UNIVERSIDAD DE ALCALÁ DE HENARES

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo de Fin de Grado

Grid de datos desarrollado con AngularJS utilizando el patrón Modelo Vista-Controlador

Mario García Salinero

Septiembre 2014

UNIVERSIDAD DE ALCALÁ DE HENARES

Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

Grid de datos desarrollado con AngularJS utilizando el patrón Modelo Vista-Controlador

Autor: Mario García Salinero

Director: Antonio Moratilla Ocaña

TRIBUNAL:

Presidente: .....

Vocal 1º:.....

Vocal 2º:.....

FECHA: .....

Dedicatoria:

A mis padres, que en todo momento han estado conmigo y me han dado la oportunidad de llegar hasta donde he llegado y me han ayudado a conseguir mis metas.

A mis amigos con los que he recorrido toda esta etapa y etapas anteriores de mi vida.

A Belinda, que en todo momento me anima y me levanta el ánimo tanto en los momentos más duros como en los más felices y que me da fuerzas para seguir cumpliendo mis sueños.

Agradecimientos:

Gracias a la Universidad de Alcalá y a sus profesores por los conocimientos que me han otorgado que me han permitido realizar este trabajo.

Agradezco a Antonio Moratilla Ocaña por guiarme a través de todo el proyecto y ofrecerme el trabajo.

Y por último, agradezco la ayuda de mis padres, de Belinda Cuesta Pérez y José Luís Martínez Pina revisando y aconsejándome mejoras.

Sumario:

Angrid es un componente software para la web que se basa en la tecnología AngularJS. Esta herramienta permitirá a un desarrollador web crear regillas para la visualización de datos con muy poco esfuerzo. El modelo Vista-Controlador utilizado por AngularJS y los estándares web utilizados para este proyecto lo hacen perfecto para el desarrollo de aplicaciones web y manejo de datos JSON.

Sumary:

Angrid is a software component for the web that is based on the AngularJS technology. This tool will allow web developer to create grids for displaying data with very little effort. The View-Controller model implemented by AngularJS and web standards used for this project make it perfect for web application development and JSON data management.

Palabras clave:

AngularJS, JSON, grid, datos, modelo vista-controlador, HTML5, CSS3, asíncrono, framework, control, web.

Resumen:

En este proyecto se describe el proceso de creación de Angrid. Angrid es un componente software para la web basado en la tecnología AngularJS.

AngularJS permite la creación de aplicaciones fundamentadas en el modelo Vista-Controlador. Así, Angrid, nace como un grid de datos basado en el modelo Vista-Controlador y en los últimos estándares web como HTML en su versión 5 y CSS 3.

El objetivo principal de Angrid es la creación de grids de datos con muy poco esfuerzo para el desarrollador y en muy pocas líneas de código. El consumo de datos JSON por parte de Angrid lo convierte en una tecnología acorde a las tendencias actuales.

Así mismo, en este proyecto, se describe el funcionamiento de AngularJS y de los distintos paradigmas y modelos de programación para la web. Se pone en entredicho antiguos arquetipos muy comúnmente usados en la web y se realiza una comparación entre estos y la tecnología utilizada por Angrid.

Índice general:

Contenido:

Tabla de ilustraciones:

1. Introducción

En la actualidad, la Web 2.0 es clave en el ámbito del internet global. Tanto en dispositivos móviles como en fijos la del usuario con la aplicación web es esencial. Esto es debido a que la Web 2.0 facilita en gran medida el uso y la eficiencia de cualquier aplicación web, además de poder producir efectos y características que llamen la atención al cliente.

La mayoría de las aplicaciones web 2.0 están diseñadas basándose en el típico modelo cliente-servidor, donde el peso de la lógica de la aplicación recae en el servidor. Uno de los grandes inconvenientes de este modelo es la carga que se le da al servidor.

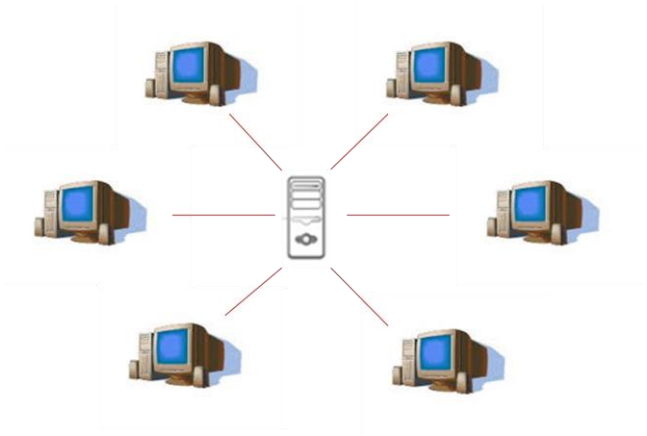


Ilustración 1 Modelo C-S

El modelo cliente-servidor se basa en una máquina servidor que procesa peticiones de multitud de máquinas cliente que reciben una respuesta final, es decir, el servidor hace el trabajo y los clientes lo muestran.

Esto es y ha sido siempre un quebradero de cabeza para muchos desarrolladores y técnicos a la hora de que un servidor pueda dar servició al mayor número de personas posible, e incluso, a la hora de ser vulnerables a ataques de usuarios maliciosos.

Otro de los inconvenientes de este modelo son las propias limitaciones de la conexión a la red, tanto por parte del servidor como del propio cliente. Dependiendo de a qué distancia se encuentre un servidor, cuánto tráfico haya en la red o qué medio físico exista entre el cliente y el servidor podremos obtener la información a mayor o menor velocidad y tiempo.

Es por eso por lo que nosotros, sin embargo, queremos tender hacia un modelo donde la lógica reside en el cliente, permitiendo el uso de nuestras aplicaciones web de manera más veloz y más independiente de nuestra conexión a internet, esto lo podemos conseguir gracias al llamado modelo vista-controlador (MVC).

El modelo vista-controlador, al contrario que el cliente-servidor, reparte el funcionamiento de una aplicación web entre el cliente, que ejecutará la lógica necesaria para generar el resultado y el servidor, que aportará al cliente solo la información necesaria para que pueda generar el mismo.

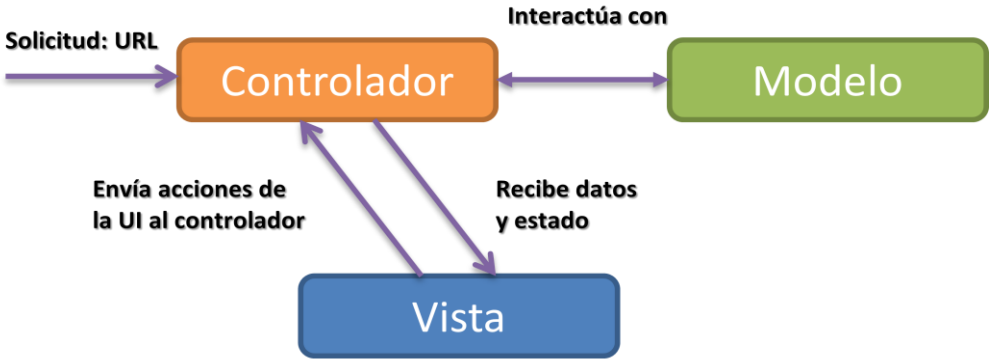


Ilustración 2 Funcionamiento MVC

Vamos a poner en práctica estos conocimientos con la creación de una aplicación web utilizando el modelo vista-controlador ya mencionado. Dispondremos, para ello, del framework AngularJS de Google para facilitarnos el trabajo. (1)

AngularJS es un framework basado en JavaScript. En cuanto a este framework lo entendemos como una estructura software, en este caso basada en el lenguaje JavaScript, con componentes personalizables e intercambiables ya existentes en el framework para desarrollar una aplicación que disponga de las facilidades y ventajas que ya están desarrolladas.

Esto nos permitirá desarrollar una aplicación con funcionalidades que ya están implementadas en AngularJS como extensión de HTML o actualización de la vista de forma transparente al usuario.

Con este concepto se deseó crear un grid de datos. Un grid nos permite visualizar, manejar, ordenar y operar con los datos de una fuente de datos. El resultado de este grid será una tabla donde cada columna representará un campo y cada fila un registro del origen de datos.

Además de generar columnas y filas para mostrar los valores de un origen de datos este grid también nos permitirá calcular y operar con el resto de campos y valores de nuestro grid y así generar y mostrar resultados de operaciones en tiempo de ejecución.

Si aplicamos este concepto de grid al concepto de AngularJS obtendremos un grid de datos que nos permitirá mostrar un origen de datos JSON, ordenarlos, paginarlos y operar desde la máquina cliente por lo que apenas podría depender de la conexión de la que disponga el usuario, la capacidad del servidor o la congestión en la red.

Incluso a la hora de actualizar la vista del cliente no será necesario recargar la página, ya que AngularJS nos permite actualizar el HTML de la página de forma dinámica haciendo más fluida la experiencia del usuario.

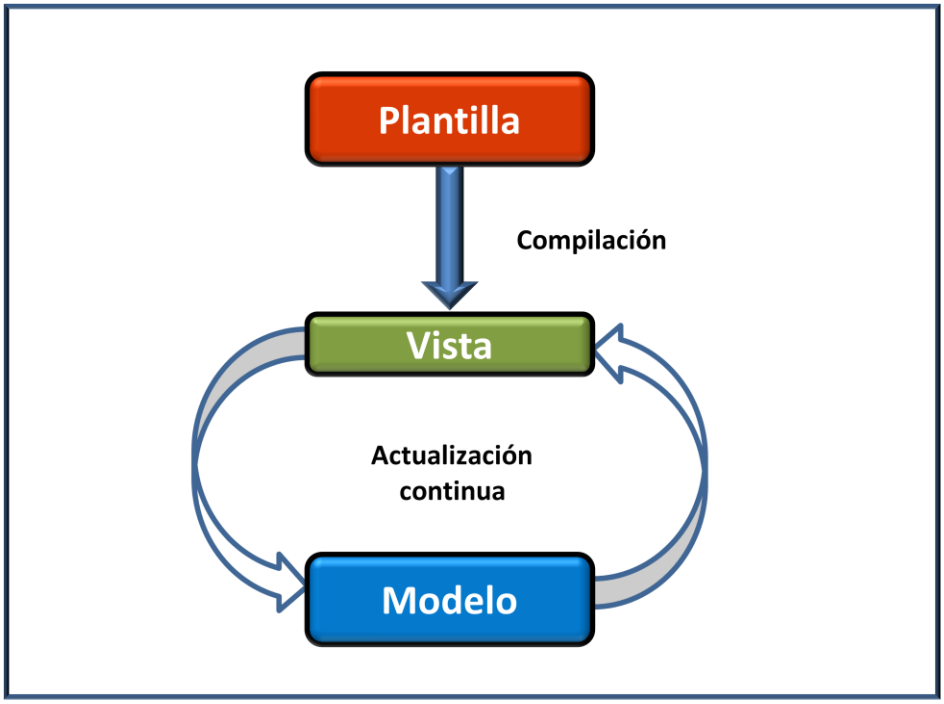


Ilustración 3 Actualización dinámica de la vista con AngularJS

Respecto al origen de datos hemos comentado que dispondremos de datos en JSON. El formato JSON es un formato ligero de intercambio de datos. Esto permite tanto a humanos como a máquinas leer, entender y convertir la información que contiene a datos útiles. Nos permitirá transferir los datos del servidor al cliente y transformarlos con mucha más velocidad.

Otra peculiaridad es que el manejo del grid deberá ser utilizando un lenguaje cliente, es decir, capaz de ser ejecutado en cualquier navegador. En este caso se ha decidido usar JavaScript ya que es el lenguaje más utilizado para esta finalidad y AngularJS está basado en él.

Gracias a JavaScript y mediante eventos y funciones abstractas definidas por el grid, se le da al desarrollador un gran abanico de posibilidades de uso y control sobre grid y los datos que contiene.

Así nace **Angrid**. Un grid que, con muy poco esfuerzo, permite al desarrollador de aplicaciones web generar rejillas de datos completamente personalizados y con un sinfín de posibilidades.

2. Objetivos

El objetivo principal de Angrid es brindar al desarrollador una manera fácil, rápida y completa de desarrollar y personalizar un grid de datos para mostrarlo en una página web. El diseño y programación en HTML de una tabla de datos para una página web, puede ser un arduo trabajo que puede requerir bastante tiempo, por eso, cada vez más se vienen utilizando paquetes, clases o frameworks que permitan al desarrollador evitar el diseño en HTML de estas engorrosas tablas.

Un desarrollador puede encontrarse con infinidad de estas herramientas codificadas en y para multitud de lenguajes para la web tanto de servidor como de cliente. Y tal es el panorama, que este tipo de recursos no es solo común en el desarrollo web, sí no que también están disponibles para otros tipos de paradigmas y plataformas de programación.

Y pese a la cantidad de estas herramientas que ya existen y se utilizan en la actualidad, como *EditableGrid* o *Ingrid*, nuestro objetivo es crear un grid de datos que supere a todas estas herramientas en cuanto a facilidad de uso, tiempo de desarrollo y capacidad de personalización.

Ilustración 4 Ejemplo de EditableGrid

Por todo ello, Angrid extiende el lenguaje HTML añadiendo nuevas etiquetas al lenguaje facilitando la creación de cualquier grid con total facilidad, gracias que un programador o diseñador de una aplicación web estará utilizando constantemente HTML a JavaScript y le será mucho más familiar que añadir complicadas funciones o sintaxis. Qué mejor manera de facilitar el desarrollo que conseguir que el grid utilice el mismo lenguaje que en el resto de la aplicación y se integre con ella como si fuese nativa. Este es uno de los objetivos más importantes de Android y lo consigue gracias a extensibilidad del lenguaje HTML con AngularJS.

Otro aspecto muy importante que se ha remarcado anteriormente, es la capacidad de personalización del grid. No debe ser necesario que el desarrollador dé formato al componente HTML, aún si bien se le debe dar la posibilidad de poder hacerlo ya que, como es de esperar,

cualquier diseñador querrá adaptar el grid al resto de su aplicación. Esto debe resultarle fácil y natural.

Del mismo modo que en la implementación del grid se utiliza HTML extendido para que al desarrollador le resulte lo más intuitiva y sencilla posible, a la hora de modificar y personalizar el estilo de una celda, la cabecera y del propio grid en si también debe serlo.

Para resolver esta situación acudimos a la manera más usual y estandarizada de definir el estilo y formato del HTML en una aplicación web, las hojas de estilos en cascadas o CSS. Los documentos CSS nos permiten definir todo el estilo de una página en un solo documento entonces, ¿por qué no definir el estilo del grid también en ese mismo documento? Efectivamente Angrid basa su formato en el estándar CSS versión 3.

```
.angrid_pageNumber {
    -webkit-transition:all linear 0.5s;
    transition:all linear 0.5s;
    background: transparent;
    width: 30px;
}

.angrid_pageNumber.ng-invalid {
    color:white;
    background: red;
}

.angrid_pageTotalNumber{
    width: 30px;
    border: none;
}
```

Ilustración 5 Ejemplo CSS de Angrid

Angrid utiliza nombres de clases CSS muy intuitivos para facilitar que el desarrollador edite sus propias hojas de estilos acorde al resto de su aplicación web, incluyendo clases con estados como *a* y *a:hover* o los propios estados de AngularJS como *ng-invalid*.

Ahora que ya hemos definido algunos objetivos básicos que debe cumplir Angrid a la hora de instanciar un grid en una aplicación web vamos a pasar al manejo del propio grid ya que no solo queremos que se cree una tabla HTML, ahora queremos que se rellene de nuestros datos.

Como se ha mencionado durante toda la etapa de este documento, Angrid va a utilizar JSON. El objetivo de utilizar JSON es que es muy fácil de leer, convertir y se puede cargar fácilmente en variables JavaScript.

La facilidad para usar JSON está impulsando este formato cada vez más popular y usado en la red. Esto quiere decir que Angrid usará este formato que cada vez más utilizado por desarrolladores y en el caso de no utilizarlo es muy fácil pasar de cualquier formato a JSON añadiendo así otra ventaja competitiva sobre otros grids que usan otros formatos como XML.

Ilustración 6 Ejemplo de datos en JSON

Para terminar, se debe dar especial hincapié en que el desarrollador pueda realizar libremente cualquier operación que se le pueda plantear, configurando o programando las columnas o el propio grid. Esto, mediante eventos y métodos JavaScript abstractos definidos por el grid, se le permitirá al desarrollador programar sus propios filtros, columnas calculadas o interacciones del usuario.

3. Estado del arte

En este apartado veremos la situación actual global en relación a este proyecto y lo compararemos con el mismo haciendo valer las ventajas que aporta Angrid.

3.1. Arquitectura

En primer lugar este proyecto se ha basado en una arquitectura de desarrollo cada vez más usual, como es la basada en el Modelo vista-controlado, como se ha visto en la introducción de este documento. Sin embargo, existen multitud de modelos y arquitecturas con las cuales se podría haber realizado este proyecto.

Uno de estas arquitecturas es el ya común Cliente-Servidor (C/S). La arquitectura C/S, aunque cada vez más va dejando paso a otros modelos en el ámbito de la web 2.0, sigue siendo la arquitectura por excelencia de la mayoría de los servicios que ofrece Internet.

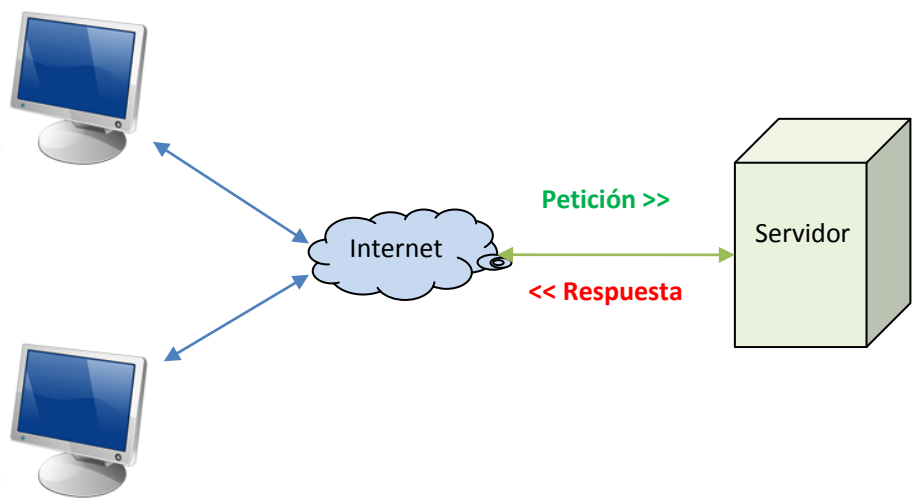


Ilustración 7 Esquema básico de C/S

Esto ya es una ventaja para C/S, ya que existen tecnologías que están suficientemente desarrolladas, diseñadas y probadas para este paradigma. Esto nos asegura que, mediante estas tecnologías, podemos realizar aplicaciones robustas, seguras, consistentes y, en definitiva, que cumplirán su cometido. (2)

Además, C/S es un paradigma que nos permite centralizar las operaciones que realizamos en un solo lugar, lo que además nos lleva ineludiblemente a que podremos mantener nuestra aplicación de manera completamente transparente al cliente. Por lo que la facilidad de mantenimiento y la centralización de las operaciones y datos dan otro punto positivo para C/S.

Aunque C/S permita centralizar los datos en un servidor no quiere decir que tengamos una limitación dada por la máquina servidora ya que C/S también nos suministra la posibilidad de escalar nuestro hardware, tanto del cliente como del servidor de manera independiente. Por lo que la escalabilidad y la heterogeneidad de tecnologías cliente que pueden conectarse le da finalmente un punto muy determinante a C/S.

Sin embargo, C/S tiene diversos inconvenientes que tropiezan con los objetivos de este proyecto y de muchos otros. El problema clave que manifiesta C/S es la congestión del tráfico y la capacidad de conexión del servidor. En este caso nos afectaría gravemente, ya que el objetivo de Angrid es mostrar los datos de la manera más fluida y rápida posible. Esto nos pondría en dificultades si muchos clientes intentasen acceder a nuestro servidor.

La congestión ha sido un problema siempre en este paradigma para todas las aplicaciones que lo usan. Es inevitable que si muchos clientes acceden al servidor constantemente al final existan problemas de conexión. Esto ha sido explotado siempre como una vulnerabilidad para usuarios malintencionados, estos ataques pueden llegar a provocar la indisponibilidad del servidor, incluso cuando son ejecutados con una baja tasa de solicitudes. (3)

Y efectivamente, la indisponibilidad es otro problema de C/S. Si el servicio al que deseamos acceder está inoperativo no podremos trabajar y el desarrollador no podrá hacer mucho para evitar este problema.

Pero, por otro lado, tenemos otro gran modelo que poco a poco se va afianzando más en la comunidad de desarrolladores web, Modelo Vista-Controlador (MVC).

El MVC nos permite realizar una segregación bien diferenciada entre lo que ve el usuario, control y actualización de la vista y la representación de los datos y lógica de funcionamiento, también llamados vista, controlador y modelo respectivamente. (4)

Gracias a este modelo la vista del usuario será independiente de la lógica de la aplicación. Esto permitirá que la actualización de la misma esté supeditada al controlador que se encargará de esto mismo, mantener las vistas actualizadas.

Esto implica una gran ventaja ya que uno de las metas de este proyecto es hacer que Angrid muestre los datos de manera más fluida, objetivo que conseguimos actualizando en tiempo de ejecución la vista sin tener que recargar la página entera.

Esta implicación nos permitirá repartir el funcionamiento entre el cliente y el servidor, reduciendo carga de trabajo al mismo.

Otra de las ventajas que vienen intrínsecamente con esta arquitectura es su modularidad. Cualquier modificación que se deba realizar en las vistas no afectará al modelo del dominio, simplemente se modificará su representación. Lo mismo sucede con los cambios en el modelo, cualquier cambio en el mismo solo implica la modificación de las interfaces con las vistas.

Como se puede observar este modelo de diseño de aplicaciones está basado en el paradigma de la programación orientada a objetos por sus características, y en efecto, así es. MVC está basado en la programación orientada a objetos y aplicar este modelo a otros paradigmas de programación complica bastante su implementación.

Asimismo, otro inconveniente destacable es el coste requerido al inicio ya que se necesita una arquitectura inicial para poder construir las diferentes partes del MVC y la propia aplicación.

Y, finalmente, además de estas arquitecturas, también podemos encontrar otros modelos como son el Modelo Vista Vista-Modelo (MVVM), en la que encontramos una variante del MVC destinado a paradigmas de programación orientada a eventos como *Silverlight* o *WPF*. (5) (6) (7)

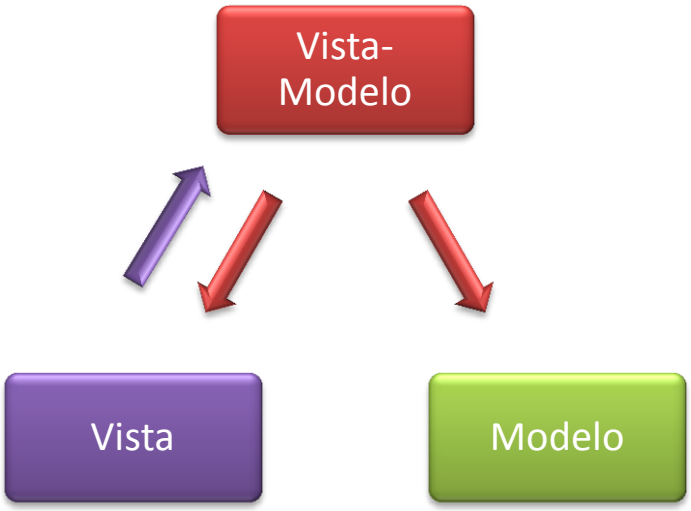


Ilustración 8 Esquema básico MVVM

Valorando todos estos modelos y la aplicación que se desea construir vemos que hay una serie de cuestiones a tener en cuenta. Queremos producir nuestra aplicación con un paradigma acorde a la tecnología JavaScript actual, es decir, orientado a objetos. También queremos que Angrid muestre los datos de manera fluida y transparente para el usuario. Y finalmente queremos que el servidor no tenga toda la carga de trabajo para evitar congestión en la red.

Claramente vemos que la arquitectura a utilizar para el proyecto de Angrid debe ser el Modelo Vista-Controlador.

3.2. Framework

A la hora de elegir como desarrollar la aplicación se han tenido en cuenta varios frameworks actuales muy importantes en la actualidad. Se han tenido en cuenta frameworks de tecnología cliente y más en concreto frameworks basados en JavaScript, una de las tecnologías cliente por excelencia en la actualidad.

(8)Uno de estos frameworks es *Blackbone.js*. Este framework está basado en otro framework JavaScript llamado *Undercore.js*. (9) Blackbone, a diferencia de su predecesor, está pensado para el MVC por lo que viene perfecto para este proyecto.

Se trata de un framework muy ligero, con una sintaxis muy parecida a Underscore y jQuery. Esto le facilita al desarrollador el aprendizaje reduciendo enormemente el coste, el tiempo y el impacto de tener que lidiar con un nuevo lenguaje.

```
var Todo = Backbone.Model.extend({

  // Default attributes for the todo item.
  defaults: function() {
    return {
      title: "empty todo...",
      order: Todos.nextOrder(),
      done: false
    };
  },

  // Toggle the `done` state of this todo item.
  toggle: function() {
    this.save({done: !this.get("done")});
  }

});
```

Ilustración 9 Ejemplo de código BlackBone

Como cualquier framework, Blackbone ofrece funciones para programar aplicaciones de manera más rápida y sencilla pero en este caso, además, nos ofrece una estructura totalmente orientada al MVC ofreciéndonos clases como *Model*, *View* o *Controller (Router).*

Gracias a esto, el programador tiene control absoluto de la estructura del código que está realizando solventando problemas de librerías prefabricadas que no nos aportan una funcionalidad concreta. Aunque esto es precisamente una gran desventaja.

Un desarrollador puede controlar toda la estructura del código de una aplicación pequeña y sencilla, sin embargo, si la aplicación aumenta de tamaño puede ser un gran quebradero de cabeza.

Por último, Blackbone es uno de los primeros frameworks JavaScript basados en el MVC para el desarrollo de *single-web apps* (aplicaciones web de una sola página) lo que lo convierte en el predecesor de otros frameworks populares como AngularJS o Ember.

Y es que, precisamente, *Ember* es otro de los frameworks JavaScript más populares de la comunidad. Su filosofía está basada en el denominado **Convention Over Configuration**. Esto es equivalente a decir "seguir las convenciones sobre configurar".

Esta filosofía permite que, con Ember, el desarrollador evite tener que programar explícitamente tareas triviales que el framework es capad de resolver. Es el llamado **Naming Conventions** que permite la generación de código implícita.

Además, la potencia de las entidades controlador (o routing) sobre el MVC de Ember hace que esté a años luz de AngularJS y Blackbone.

Todo esto permite al desarrollador programar "a la manera Ember" (*The Ember Way*), lo que se traduce en realizar las operaciones de forma fácil, rápida y con muy pocas líneas de código.

Esto se consigue gracias a su filosofía "Convention Over Configuration" mencionada anteriormente. Sin embargo, en el momento en el que el desarrollador desea realizar algo que se sale de la convención o de la norma establecida la elegancia, facilidad y sencillez de Ember desaparecen por completo.

Esta estructura tan rígida incluye otros graves inconvenientes como la dificultad de aprendizaje, ya que sin conocer completamente el funcionamiento del framework será complicado para un desarrollador programar usando Ember.

Todo esto, unido a la documentación poco actualizada que existe actualmente y a los cambios frenéticos que está sufriendo, hace que Ember quede descartada por el momento como el framework elegido.

Y finalmente, el novedoso framework de Google, AngularJS. Este framework que ha aparecido hace relativamente pronto se ha impuesto con fuerza. Tal es así que, en búsquedas sobre información sobre él supera en cinco veces a sus competidores.

Y es que no es para menos, ya que el sistema que utiliza AngularJS es completamente novedoso ya que su filosofía se basa en la extensión del código HTML.

La comunidad de usuarios de AngularJS ha crecido exponencialmente desde su lanzamiento y junto con el soporte que brinda Google, hace que el aprendizaje sea más fácil, pese a que la curva de aprendizaje del propio framework sea algo abrupta.

Directivas, servicios, proveedores, factorías; son algunos de los conceptos esenciales en AngularJS que no son fáciles de entender hasta que no has tenido bastante contacto con el framework.

En cuanto a ventajas, la extensión del HTML es una acertadísima manera de facilitar al desarrollador final la programación de aplicaciones. En este caso, utilizando unas simples etiquetas HTML con sus atributos se podría añadir un grid Angrid en la web.

AngularJS facilita enormemente el diseño de las aplicaciones que adoptan un esbozo fluido y responsive, aplicando el MVC de una forma muy especial.

Al contrario que otros framewors, AngularJS actualiza continuamente la vista y, utilizando los denominados $watches escucha los cambios en el DOM realizados por el usuario y se encara de recompilarlo.

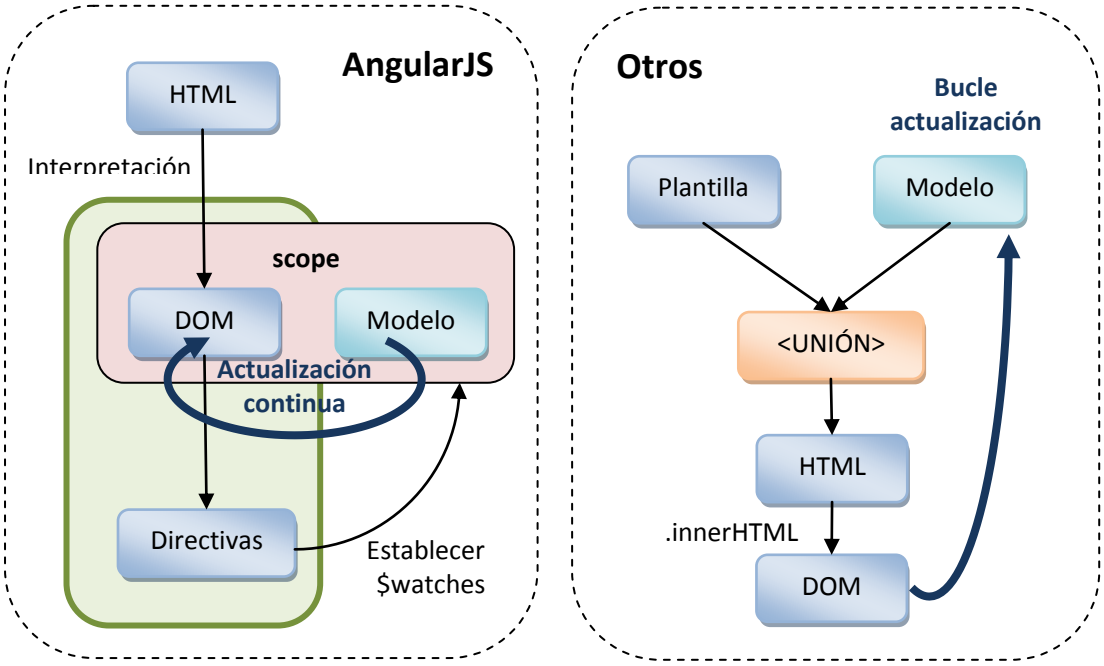


Ilustración 10 Comparativa de actualización del DOM

Después de ver ventajas e inconvenientes de estos y otros frameworks la decisión de utilizar AngularJS para el desarrollo de Angrid es más que correcta. La extensión del lenguaje HTML permitirá que el desarrollador no sienta que está usando un framework y le parecerá que su web solo usa HTML.

El código final de Angrid, por tanto, será sencillo, entendible y hará que otros desarrolladores sin conocimientos de AngularJS lo utilicen.

Este framework cumple con varios objetivos que teníamos marcados para este proyecto; como son la fluidez, facilidad de uso, facilidad de mantenimiento y facilidad de legibilidad del código.

3.3. Dificultad en el desarrollo

Uno de los aspectos más importantes de este software es la facilidad que se le dá al programador para desarrollar el grid a su gusto.

En este sentido podemos comparar diferentes grids existentes en el mercado y analizar la complejidad final que utilizaría cada uno.

4. Definición del sistema

4.1. Descripción del problema

El problema se basa en desarrollar un grid utilizando AngularJS, JavaScript y HTML5. Para ello, se desean desarrollar unas clases para el control y la instanciación del grid.

Debe ser capad de manejar todos los tipos de datos posibles y su entrada debe ser en formato JSON o entrada directa del usuario en el propio grid.

El grid no tiene que acceder a la base de datos pero si debe dar soporte a que una función JavaScript pueda devolver el JSON u obtener los datos directamente de un archivo.

Debe ser también altamente personalizable, permitiendo al desarrollador acceder a funciones para el manejo del grid y crear funciones JavaScript personalizadas para las columnas y filas del grid. También es imprescindible que el desarrollador pueda generar las columnas y las filas fácilmente y sin necesidad de ser un experto en tecnologías que no sean HTML o JavaScript.

Finalmente se ha postulado esta aplicación para utilizarlo con el MVC y sin necesidad de actualizar la página por parte del usuario.

4.2. Descripción de la solución

Se ha decidido realizar un grid con AngularJS que nos permite extender el lenguaje HTML para facilitar la comprensión léxica para desarrollador.

Se utilizará una estructura jerárquica, como lo es HTML, para definir la forma del grid.

Para ello, los roles determinados que actúan en la aplicación son el desarrollador y el propio grid.

4.3. Requisitos

4.3.1. Requisitos funcionales

Código	Nombre	Descripción	Prioridad
RF01	Formato de entrada	La entrada de datos utilizará el formato JSON.	ALTA
RF02	Entrada desde archivo	El JSON se podrá obtener de un fichero.	BAJA
RF03	Entrada desde memoria	El JSON se podrá obtener desde una variable JavaScript cargada en memoria.	ALTA
RF04	Uso de AngularJS	El grid tendrá que ser definido como una aplicación AngularJS.	ALTA

RF05	HTML extendido	Se usará el lenguaje HTML para definir el grid.	ALTA
RF06	Etiqueta ANGRID	La etiqueta principal para definir el grid será <ANGRID>.	BAJA
RF07	Definición de la entrada desde archivo	Se utilizará el atributo *filesource* en la etiqueta *ANGRID* junto con la ruta del archivo JSON para cargarlo.	BAJA
RF08	Definición de la entrada desde variable	Se utilizará el atributo *source* en la etiqueta *ANGRID* junto con el nombre de la función JavaScript que devolverá la variable con el JSON.	MEDIA
RF09	Prioridad de entradas	Si se indicasen los atributos *source* y *filesource* en la etiqueta *ANGRID* prevalecería la entrada por variable.	BAJA
RF10	Paginación	Si se indica el atributo *paginated* en la etiqueta *ANGRID* el grid será paginado.	MEDIA
RF11	Elementos por página	Los elementos en cada página vendrán determinados por el atributo *paginated* en la etiqueta *ANGRID.*	MEDIA
RF12	Página actual	Se podrá definir la página inicial con el atributo *currentpage* de la etiqueta *ANGRID.*	BAJA
RF13	Requisitos del grid	La etiqueta *ANGRID* debe tener el atributo *source* o *filesource* de manera requerida.	ALTA
RF14	Eventos	Todos los atributos que representen un evento se les deberá asociar el nombre de la función JavaScript sin paréntesis que será llamada al producirse dicho evento.	ALTA
RF15	Evento onLoad	La etiqueta *ANGRID* podrá tener un evento que se lanzará al completar el procesamiento de la directiva.	MEDIA
RF16	Parámetros onLoad	El evento *onLoad* no tendrá parámetros.	BAJA
RF17	Evento onPageChange	La etiqueta *ANGRID* podrá tener un evento que se lanzará al cambiar de página.	ALTA
RF18	Parámetros onPageChange	El evento *onPageChange* se le pasará la página nueva a la que se ha cambiado.	ALTA
RF19	Evento onCompileStart	La etiqueta *ANGRID* podrá tener un evento que se lanzará después de generar el HTML que se compilará.	BAJA

RF20	Parámetros onCompileStart	El evento *onCompileStart* pasará como parámetro el HTML que se compilará.	BAJA
RF21	Evento onCompile	La etiqueta *ANGRID* podrá tener un evento que se lanzará justo después de compilar el HTML.	BAJA
RF22	Parámetros onCompile	El evento *onCompile* pasará como parámetro el scope actual de la compilación.	BAJA
RF23	Evento onCompileComplete	La etiqueta *ANGRID* podrá tener un evento que se lanzará después de compilar y añadir el código al DOM y de ejecutar los scripts de las columnas.	MEDIA
RF24	Parámetros onCompileComplete	El evento *onCompileComplete* pasará como parámetro la fachada generada para esa instancia del grid.	MEDIA
RF25	Etiqueta COLUMN	El grid contará con una etiqueta COLUMN para definir cada una de las columnas de la tabla.	ALTA
RF26	Posición COLUMN	La etiqueta COLUMN deberá ser hija de la etiqueta ANGRID.	ALTA
RF27	Nombre de la columna	A cada columna se le asignará un nombre mediante el atributo *name*.	MEDIA
RF28	Nombre de columna válido	El atributo *name* no debe ser un número.	BAJA
RF29	Nombre para mostrar	El texto que se mostrará como representación de la columna será el mismo que el nombre de la columna.	MEDIA
RF30	Fuente de la columna	La columna puede tener origen en el JSON de entrada, esto se define mediante el atributo *source* de la etiqueta COLUMN.	ALTA
RF31	Script de la columna	Se puede asignar un nombre de función JavaScript para cada columna que se ejecutará para cada elemento de la columna justo antes de *onCompileComplete* mediante el atributo *script.*	ALTA
RF32	Parámetros de los scripts	A las funciones JavaScript de los atributos *script* se les pasarán el ID de la celda, el valor que contiene la misma, el número de columna en la que se encuentra, el número de fila, el nombre de la columna y la fachada.	ALTA
RF33	Columna oculta	El atributo *hidden* de la etiqueta columna permite	MEDIA

		ocultar dicha columna al usuario pero será accesible desde JavaScript.	
RF34	Columnas editables	Se podrá interactuar en el contenido de las celdas de esa columna si la columna se marca como *editable.*	ALTA
RF35	Columnas ordenables	Podrá ordenarse el contenido de la columna si se marca la misma como *sortable*.	MEDIA
RF36	Tipos de columnas	Se podrá modificar el tipo de dato y su representación en cada columna indicándolo en el atributo *type.*	BAJA
RF37	Columnas checkbox	Las columnas que se marquen como *type=check* contendrán un checkbox y su activación viene dada por su valor.	MEDIA
RF38	Checkbox editables	Si las columnas de tipo checkbox son editables podrán marcarse y desmarcase sus valores.	MEDIA
RF39	Columnas imagen	Las columnas que se marquen como *type=image* contendrán una imagen cuya URL viene determinada por el valor de la celda.	BAJA
RF40	Imágenes editables	El valor editable no afectará a las columnas de tipo imagen.	BAJA
RF41	Columnas lista	Las columnas que se marquen como *type=list* contendrán un combo box cuyo elemento seleccionado será el valor de la celda.	MEDIA
RF42	Listas editables	El valor editable permitirá cambiar la selección de la lista.	ALTA
RF43	Items de la lista	Los ítems de la lista vienen determinados por el atributo *ítems*. Contiene un nombre de función que devuelve un JSON con los elementos.	ALTA
RF44	Columna botón	Las columnas que se marquen como *type=column* contendrán un botón cuyo texto será el valor de la celda.	BAJA
RF45	Columna texto	Las columnas que no tengan tipo o se marquen como otro tipo distinto serán columnas con texto plano o HTML.	ALTA
RF46	Textos editables	El atributo editable permitirá modificar el contenido textual de una celda.	ALTA

RF47	Estilo del grid	Se pueden asignar clases de hojas de estilos mediante el atributo *class* a todos los elementos del grid.	ALTA
RF48	Tamaño imágenes	El tamaño de las imágenes puede determinarse mediante los atributos *width* y *height.*	BAJA
RF49	Imagen predeterminada	En las columnas de tipo imagen que no tengan atributo *source* se les puede determinar la ruta de una imagen mediante *imagesource*.	MEDIA
RF50	Checkboxes activados	En las columnas de tipo checkbox que no tengan atributo source se les puede establecer su valor con el atributo *checked.*	MEDIA
RF51	OnClick	Todas los tipos de columna reaccionan ante el evento *onClick* en cuyo atributo se deberá insertar el nombre de una función JavaScript que maneje el evento.	ALTA
RF52	Parámetros onClick	Se le pasará el ID y la fachada a la función JavaScript manejadora de *onClick*.	ALTA
RF53	OnChange	Las columnas de tipo lista lanzarán el evento *onChange* cuando cambie la selección.	ALTA
RF54	Parámetros onChange	Se le pasará el ID y la fachada a la función JavaScript manejadora de *onChange*.	ALTA
RF55	Fuente y script de columna	Una columna puede tener el atributo *source*, *script* o ambos.	ALTA
RF56	Requerimientos de columna	La etiqueta COLUMN requiere el atributo *name* y éste debe ser único en todo el grid y no debe ser un número.	MEDIA
RF57	Control de paginado	La etiqueta *CONTROLPAGINA* añade un control en la posición en la que se ponga que permite pasar las páginas del grid.	MEDIA
RF58	Contenido del control de páginas	Dispondrá de botón de inicio, botón de atrás, página X de N, botón de siguiente y botón de fin respectivamente.	MEDIA
RF59	Cambio de página	Se podrá cambiar de página usando los botones de atrás, siguiente, inicio y fin o escribiendo el número de página en el cuadro de texto.	MEDIA
RF60	Página válida	Si la página introducida en el cuadro de texto no es	BAJA

		válida se hará notar al usuario en color rojo.	
RF61	Extender contenido	El contenido de la etiqueta *CONTROLPAGINA* se añadirá a continuación del control de páginas.	BAJA
RF62	Inicio de páginas	Las páginas comenzarán en 1.	BAJA
RF63	Fachada	Se debe definir una fachada con la que el usuario pueda interactuar con todo el grid.	ALTA
RF64	Acceso al grid	Se puede acceder a la clase grid desde la fachada utilizando *fachada.grid.*	BAJA
RF65	Acceso al modelo	Se puede acceder al modelo de datos JSON desde la fachada utilizando *fachada.datos.*	ALTA
RF66	Acceso al scope	Se puede acceder al scope del DOM desde la fachada usando *fachada.scope.*	BAJA
RF67	Actualizar vista	Utilizando *fachada.updateView* se actualizan los scripts de la vista del grid actual.	BAJA
RF68	Actualizar DOM	Se recompila el DOM utilizando *fachada.updateDOM.*	BAJA
RF69	Actualizar todo	Se actualiza el DOM y la vista desde la fachada utilizando el método *update* de la misma.	ALTA
RF70	Obtener valor de la vista	Obtiene el valor de una celda en la vista actual mediante el método *getCellValue* de la fachada.	ALTA
RF71	Establecer valor de la vista	Establece el valor de una celda en la vista actual mediante el método *setCellValue* de la fachada.	ALTA
RF72	Obtener una celda	Obtiene el elemento HTML de una celda de la vista actual con el método *getCellElement* de la fachada.	MEDIA
RF73	Obtener valor del modelo	Obtiene el valor de una celda en el modelo de datos actual mediante el método *getDataValue* de la fachada.	ALTA
RF74	Establecer el valor del modelo	Establece el valor de una celda en el modelo de datos actual mediante el método *setDataValue* de la fachada.	ALTA
RF75	Obtener columna por ID	Obtiene el número de columna en la que se encuentra una celda mediante su ID con la fachada.	MEDIA
RF76	Obtener fila por ID	Obtiene el número de fila en la que se encuentra una celda mediante su ID.	MEDIA
RF77	Obtener número de filas	Obtener el número de filas del modelo actual.	MEDIA

RF78	Obtener número de columnas	Obtener el número de columnas del grid actual.	BAJA
RF79	Obtener el tipo de columna	Obtiene el tipo de dato que se está representando marcado por el atributo *type* de la columna.	BAJA
RF80	Establecer el tipo de columna	Establecer el tipo de dato que se está representando en tiempo de ejecución.	MEDIA
RF81	Obtener el ID único de una fila	Obtiene el identificador HashID que otorga AngularJS con *repeat*.	BAJA
RF82	Establecer la página actual	Establece la página actual que se está mostrando y lanzar el evento correspondiente.	MEDIA
RF83	Obtener página actual	Obtiene la página actual que se está mostrando	BAJA
RF84	Establecer elementos por página	Establecer el número de elementos que se mostrará en cada página.	MEDIA
RF85	Obtener elementos por página	Obtener el número de elementos que se mostrará en cada página.	BAJA
RF86	Establecer paginación	Establece si se debe paginar el grid.	ALTA
RF87	Obtener paginación	Obtiene si el grid se está paginando.	BAJA
RF88	Cambiar la fuente de datos	Cambiar la fuente de datos en tiempo de ejecución a otra fuente de datos mediante un script.	ALTA

4.3.2. Requisitos no funcionales

Código	Nombre	Descripción	Prioridad
RNF01	Disponibilidad	Debe estar disponible con y sin conexión.	ALTA
RNF02	Portabilidad	Debe poder adecuarse a cualquier sistema cliente y su funcionalidad principal compatible con los navegadores estándar.	MEDIA
RNF03	Usabilidad	Debe pode ser fácil de utilizar para el programador aportando un método de manejo conocido (HTML).	ALTA
RNF04	Interoperabilidad	Debe ser altamente interoperable con JavaScript en llamadas a métodos y eventos.	ALTA
RNF05	Mantenibilidad	Debe ser fácilmente mantenible para futuros desarrollos tanto el grid como las aplicaciones cliente.	BAJA

4.4. Casos de uso

Nombre caso de uso:	**Crear grid desde archivo**
Descripción:	El usuario crea un grid desde un archivo JSON definiendo la aplicación AngularJS y usando la etiqueta ANGRID.
Lista de requisitos cubiertos:	• Formato de entrada • Entrada desde archivo • AngularJS • HTML extendido • Etiqueta ANGRID • Definición de la entrada desde archivo • Requisitos del grid

Nombre caso de uso:	**Crear grid desde memoria**
Descripción:	El usuario crea un grid desde un JSON en una función JavaScript definiendo la aplicación AngularJS y usando la etiqueta ANGRID.
Lista de requisitos cubiertos:	• Formato de entrada • Entrada desde variable • AngularJS • HTML extendido • Etiqueta ANGRID • Definición de la entrada desde variable • Requisitos del grid

Nombre caso de uso:	**Crear grid paginado**
Descripción:	El usuario crea un grid que pagina los datos de forma automática.
Lista de requisitos cubiertos:	• Paginación • Elementos por página • Página actual

Nombre caso de uso:	**Manejar eventos carga**
Descripción:	El usuario puede realizar funciones al compilar el grid.

Lista de requisitos cubiertos:	• Evento • Evento onLoad • Parámetros onLoad

Nombre caso de uso:	**Manejar eventos de compilación**
Descripción:	El usuario puede realizar funciones al compilar el grid.
Lista de requisitos cubiertos:	• Evento • Evento onCompileStart • Evento onCompile • Evento onCompileComplete • Parámetros onCompileStart • Parámetros onCompile • Parámetros onCompileStart • Fachada

Nombre caso de uso:	**Manejar eventos de paginado**
Descripción:	El usuario puede realizar funciones cuando se cambia de página en el grid.
Lista de requisitos cubiertos:	• Evento • Evento onPageChange • Parámetros onPageChange

Nombre caso de uso:	**Crear columna de tipo texto**
Descripción:	El usuario crea una columna cuyo contenido será un texto.
Lista de requisitos cubiertos:	• HTML extendido • Etiqueta COLUMN • Posición COLUMN • Nombre de la columna • Nombre de columna válido • Nombre para mostrar • Tipos de columnas • Columnas texto

Nombre caso de uso:	**Crear columna de texto editable**
Descripción:	El usuario crea una columna cuyo contenido será un texto que se podrá modificar en tiempo de ejecución.
Lista de requisitos cubiertos:	• Columnas editables • Textos editables

Nombre caso de uso:	**Crear columna con un checkbox**
Descripción:	El usuario crea una columna de tipo *check* cuyo contenido será un checkbox.
Lista de requisitos cubiertos:	• HTML extendido • Etiqueta COLUMN • Posición COLUMN • Nombre de la columna • Nombre de columna válido • Nombre para mostrar • Tipos de columnas • Columnas checkbox

Nombre caso de uso:	**Crear columna con checkbox cambiable**
Descripción:	El usuario crea una columna de tipo *check* con un checkbox que cambia su estado al hacerle click.
Lista de requisitos cubiertos:	• Columnas editables • Checkbox editables

Nombre caso de uso:	**Checkboxes ya activados en las columnas**
Descripción:	El usuario puede establecer el valor del checkbox sin *source*.
Lista de requisitos cubiertos:	• Checkboxes activados

Nombre caso de uso:	**Crear columnas con imágenes**

Descripción:	El usuario crea una columna cuyo contenido será la URL de una imagen que se mostrará.
Lista de requisitos cubiertos:	• HTML extendido • Etiqueta COLUMN • Posición COLUMN • Nombre de la columna • Nombre de columna válido • Nombre para mostrar • Tipos de columnas • Columnas imagen

Nombre caso de uso:	**Cargar imágenes en una columna**
Descripción:	El usuario crea una columna cuya imagen no está determinada por los datos de entrada.
Lista de requisitos cubiertos:	• Imagen predeterminada

Nombre caso de uso:	**Establecer el tamaño de las imágenes**
Descripción:	El usuario establece manualmente el tamaño de las imágenes.
Lista de requisitos cubiertos:	• Tamaño imágenes

Nombre caso de uso:	**Crear columnas con listas**
Descripción:	El usuario crea una columna cuyo control sea la selección de un elemento de una lista.
Lista de requisitos cubiertos:	• HTML extendido • Etiqueta COLUMN • Posición COLUMN • Nombre de la columna • Nombre de columna válido • Nombre para mostrar • Tipos de columnas • Columnas lista • Items de la lista

Nombre caso de uso:	**Columnas con listas desactivadas**
Descripción:	El usuario desactiva la selección de la lista usando el parámetro *editable.*
Lista de requisitos cubiertos:	• Listas editables

Nombre caso de uso:	**Manejar el cambio de selección de la lista**
Descripción:	Usar el evento onChange para manejar cuándo cambie la selección de la lista de una celda.
Lista de requisitos cubiertos:	• Evento • OnChange • Parámetros OnChange

Nombre caso de uso:	**Crear columnas con botones**
Descripción:	El usuario crea una columna cuyo contenido serán botones con el texto del modelo de datos.
Lista de requisitos cubiertos:	• HTML extendido • Etiqueta COLUMN • Posición COLUMN • Nombre de la columna • Nombre de columna válido • Nombre para mostrar • Tipos de columnas • Columna botón

Nombre caso de uso:	**Manejar el click en las columnas**
Descripción:	El usuario puede definir la función que manejará el evento de hacer click sobre una celda de una columna.
Lista de requisitos cubiertos:	• Evento • OnClick • Parámetros OnClick

Nombre caso de uso:	**Columnas con script**
Descripción:	El usuario puede definir una función que se ejecutará para todas las filas de una columna.
Lista de requisitos cubiertos:	• Fachada

Nombre caso de uso:	**Formato del grid**
Descripción:	El usuario puede añadir hojas de estilos y asignarles clases de estilos CSS a los elementos del grid utilizando el atributo *class.*
Lista de requisitos cubiertos:	• Estilo del grid

Nombre caso de uso:	**Ordenación del grid**
Descripción:	Las columnas que tengan el atributo *sortable* podrán ser clave para la ordenación de los datos.
Lista de requisitos cubiertos:	• Columnas ordenables

Nombre caso de uso:	**Control de paginado**
Descripción:	El usuario puede crear el control de paginado usando la etiqueta *CONTROLPAGINA.*
Lista de requisitos cubiertos:	• HTML extendido • Control de paginado • Contenido del control de páginas • Página válida

Nombre caso de uso:	**Control del grid a bajo nivel**
Descripción:	El usuario puede acceder a la clase del grid utilizando la fachada.
Lista de requisitos cubiertos:	• Acceso al grid

Nombre caso de uso:	**Acceso al modelo**
Descripción:	El usuario puede obtener el modelo de datos actualmente cargado en formato JSON a través de la fachada.
Lista de requisitos cubiertos:	• Acceso al modelo

Nombre caso de uso:	**Acceso al scope**
Descripción:	El usuario puede acceder al scope AngularJS del DOM del grid.
Lista de requisitos cubiertos:	• Acceso al scope

Nombre caso de uso:	**Actualizar**
Descripción:	El usuario puede actualizar la vista, el modelo o ambos.
Lista de requisitos cubiertos:	• Actualizar vista • Actualizar DOM • Actualizar

Nombre caso de uso:	**Obtener contenido de una celda del grid**
Descripción:	Permite obtener el objeto HTML de una celda.
Lista de requisitos cubiertos:	• Obtener una celda

Nombre caso de uso:	**Obtener HTML de una celda del grid**
Descripción:	Permite obtener el valor en HTML de una celda de la tabla.
Lista de requisitos cubiertos:	• Obtener valor de la vista

Nombre caso de uso:	**Establecer HTML de una celda del grid**

Descripción:	Permite establecer el valor HTML de una celda de la tabla.
Lista de requisitos cubiertos:	• Establecer valor de la vista

Nombre caso de uso:	**Obtener datos de una celda**
Descripción:	Permite obtener el valor de la fuente de datos relacionado a una celda.
Lista de requisitos cubiertos:	• Obtener valor del modelo

Nombre caso de uso:	**Establecer datos de una celda**
Descripción:	Permite establecer el valor de la fuente de datos relacionado a una celda.
Lista de requisitos cubiertos:	• Establecer valor del modelo

Nombre caso de uso:	**Buscar columna de una celda**
Descripción:	Permite obtener el número de columna en la que esta la celda.
Lista de requisitos cubiertos:	• Obtener columna por ID

Nombre caso de uso:	**Buscar fila de una celda**
Descripción:	Permite obtener el número de fila en la que está una celda.
Lista de requisitos cubiertos:	• Obtener fila por ID

Nombre caso de uso:	**Obtener número de filas**
Descripción:	Obtiene la cantidad del filas existentes en el modelo.

Lista de requisitos cubiertos:	• Obtener número de filas

Nombre caso de uso:	**Obtener número de columnas**
Descripción:	Obtiene la cantidad del columnas existentes en el grid.
Lista de requisitos cubiertos:	• Obtener número de columnas

Nombre caso de uso:	**Obtener tipo de columna**
Descripción:	Obtiene el texto identificativo del tipo de una columna.
Lista de requisitos cubiertos:	• Obtener el tipo de columna

Nombre caso de uso:	**Establecer tipo de columna**
Descripción:	Establece el tipo de la columna seleccionada.
Lista de requisitos cubiertos:	• Establecer el tipo de columna

Nombre caso de uso:	**Obtener el ID único de fila**
Descripción:	Obtiene el HashID único asignado por AngularJS.
Lista de requisitos cubiertos:	• Obtener el ID único de una fila

Nombre caso de uso:	**Establecer la página actual**
Descripción:	Establece el número de página que se mostrará actualmente.
Lista de requisitos cubiertos:	• Establecer la página actual

Nombre caso de uso:	**Obtener la página actual**
Descripción:	Obtiene el número de página que se mostrará actualmente.
Lista de requisitos cubiertos:	• Obtener la página actual

Nombre caso de uso:	**Establecer elementos por página**
Descripción:	Establece el número de elementos que se mostrarán en cada página del grid paginado.
Lista de requisitos cubiertos:	• Establecer elementos por página

Nombre caso de uso:	**Obtener elementos por página**
Descripción:	Obtiene el número de elementos que se muestran en cada página del grid paginado.
Lista de requisitos cubiertos:	• Obtener elementos por página

Nombre caso de uso:	**Establecer paginación**
Descripción:	Establece si se debe paginar el grid.
Lista de requisitos cubiertos:	• Establecer paginación

Nombre caso de uso:	**Obtener paginación**
Descripción:	Obtiene si se está paginando el grid.
Lista de requisitos cubiertos:	• Obtener paginación

Nombre caso de uso:	**Cambiar la fuente de datos**

Descripción:	Cambia el modelo de datos actual del grid por otro.
Lista de requisitos cubiertos:	• Cambiar la fuente de datos

4.5. Plan de desarrollo software

Caso de uso	Requisito asociado	Comentarios	Prioridad	Riesgo
Crear grid desde archivo	• Formato de entrada • Entrada desde archivo • AngularJS • HTML extendido • Etiqueta ANGRID • Definición de la entrada desde archivo • Requisitos del grid	Se debe permitir al programador cargar los datos del grid desde un archivo.	**Bajo**	**Bajo**
Crear grid desde memoria	• Formato de entrada • Entrada desde variable • AngularJS • HTML extendido • Etiqueta ANGRID • Definición de la entrada desde variable • Requisitos del grid	Se le debe permitir al programador cargar los datos del grid desde una variable JavaScript.	**Alto**	**Alto**
Crear grid paginado	• Paginación • Elementos por página • Página actual	El programador podrá paginar el grid mediante el uso de el atributo *paginated.*	**Medio**	**Bajo**
Manejar eventos carga	• Evento • Evento onLoad • Parámetros onLoad	Podrá saber cuándo se ha cargado el grid en el DOM.	**Bajo**	**Bajo**
Manejar eventos de compilación	• Evento • Evento onCompileStart • Evento onCompile • Evento onCompileComplete • Parámetros onCompileStart • Parámetros onCompile • Parámetros onCompileStart	Podrá manejar los distintos pasos de compilación del contenido del grid.	**Medio**	**Bajo**

	• Fachada			
Manejar eventos de paginado	• Evento • Evento onPageChange • Parámetros onPageChange	Podrá saber y manejar cuándo cambia de página el grid.	**Alto**	**Bajo**
Crear columna de tipo texto	• HTML extendido • Etiqueta COLUMN • Posición COLUMN • Nombre de la columna • Nombre de columna válido • Nombre para mostrar • Tipos de columnas • Columnas texto	Podrá crear columnas cuyas celdas contengan objetos de tipo texto.	**Alto**	**Alto**
Crear columna de texto editable	• Columnas editables • Textos editables	Las columnas de tipo texto podrán cambiar el valor en tiempo de ejecución por interacción del usuario.	**Medio**	**Alto**
Crear columna con un checkbox	• HTML extendido • Etiqueta COLUMN • Posición COLUMN • Nombre de la columna • Nombre de columna válido • Nombre para mostrar • Tipos de columnas • Columnas checkbox	Podrá establecer que el contenido de las celdas de una columna serán controles de tipo checkbox que estarán activas o no según el contenido del modelo de datos.	**Bajo**	**Bajo**
Crear columna con checkbox cambiable	• Columnas editables • Checkbox editables	Se deberá implementar la posibilidad de que el programador pueda establecer que el usuario pueda cambiar el valor de las checkboxes en tiempo de ejecución.	**Medio**	**Bajo**
Checkboxes ya activados en las columnas	• Checkboxes activados	Los checkboxes podrán ser activados por el programador al inicio mediante los atributos de las etiquetas *column*	**Bajo**	**Bajo**
Crear columnas con	• HTML extendido • Etiqueta COLUMN	Podrá crear columnas cuyo contenido son	**Bajo**	**Bajo**

imágenes	• Posición COLUMN • Nombre de la columna • Nombre de columna válido • Nombre para mostrar • Tipos de columnas • Columnas imagen	imágenes a través de una URL de imagen determinada por el contenido del modelo de datos.		
Cargar imágenes en una columna	• Imagen predeterminada	Podrá seleccionar una imagen predeterminada independiente del modelo de datos.	**Bajo**	**Bajo**
Establecer el tamaño de las imágenes	• Tamaño imágenes	Podrá seleccionar un ancho y un alto específico para las imágenes que contengan las columnas de tipo imagen.	**Bajo**	**Bajo**
Crear columnas con listas	• HTML extendido • Etiqueta COLUMN • Posición COLUMN • Nombre de la columna • Nombre de columna válido • Nombre para mostrar • Tipos de columnas • Columnas lista • Items de la lista	Podrá crear columnas cuyo contenido será un control de lista desplegable cuyo contenido viene dado por el valor en el modelo de datos.	**Bajo**	**Bajo**
Columnas con listas desactivadas	• Listas editables	Editando el atributo *editable* el usuario no podrá seleccionar los elementos de la lista.	**Bajo**	**Bajo**
Manejar el cambio de selección de la lista	• Evento • OnChange • Parámetros OnChange	El programador podrá manejar cuándo cambia el valor seleccionado de la lista.	**Medio**	**Bajo**
Crear columnas con botones	• HTML extendido • Etiqueta COLUMN • Posición COLUMN • Nombre de la columna • Nombre de columna válido • Nombre para mostrar • Tipos de columnas	Podrá crear columna cuyo contenido sean botones. El texto del botón vendrá determinado por el valor del modelo de datos.	**Bajo**	**Bajo**

	• Columna botón			
Manejar el click en las columnas	• Evento • OnClick • Parámetros OnClick	Se podrá manejar el evento de click sobre cualquier columna del grid.	**Medio**	**Alto**
Columnas con script	• Fachada	Se le puede añadir a las columnas una función que se ejecutará para cada celda de la columna.	**Alto**	**Alto**
Formato del grid	• Estilo del grid	Se pueden aplicar clases de estilos CSS3 mediante el atributo class en el elemento.	**Alto**	**Bajo**
Ordenación del grid	• Columnas ordenables	Se podrá ordenar el grid por las columnas que tengan el atributo *sortable*.	**Medio**	**Alto**
Control de paginado	• HTML extendido • Control de paginado • Contenido del control de páginas • Página válida	Puede añadir un control para que el usuario final pueda cambiar de página.	**Medio**	**Medio**
Control del grid a bajo nivel	• Acceso al grid	Podrá acceder a la clase del grid para añadir nuevas columnas, recompilar, etc.; usando la fachada.	**Bajo**	**Alto**
Acceso al modelo	• Acceso al modelo	Puede acceder al modelo de datos que está utilizando actualmente el grid mediante la fachada.	**Alto**	**Alto**
Acceso al scope	• Acceso al scope	El programador puede acceder al scope de AngularJS del grid mediante la fachada.	**Bajo**	**Alto**
Actualizar	• Actualizar vista • Actualizar DOM • Actualizar	Podrá actualizar el grid mediante los métodos de la fachada.	**Alto**	**Medio**
Obtener contenido de una celda del grid	• Obtener una celda	Podrá el contenido de una celda como HTMLElement.	**Bajo**	**Alto**
Obtener HTML de una celda del grid	• Obtener valor de la vista	Podrá obtener el código HTML que contiene una celda.	**Alto**	**Alto**
Establecer	• Establecer valor de	Podrá establecer el	**Alto**	**Alto**

HTML de una celda del grid	la vista	código HTML de una celda.		
Obtener datos de una celda	• Establecer valor del modelo	Podrá obtener el valor del modelo de datos en una celda.	**Alto**	**Alto**
Establecer datos de una celda	• Establecer valor del modelo	Podrá establecer el valor del modelo de una celda.	**Alto**	**Alto**
Buscar columna de una celda	• Obtener columna por ID	Permitirá obtener el número de columna que corresponde a una celda.	**Medio**	**Alto**
Buscar fila de una celda	• Obtener fila por ID	Permitirá obtener el número de fila que ocupa una celda.	**Bajo**	**Medio**
Obtener el número de filas	• Obtener número de filas	Permitirá obtener el número de filas que ocupa el modelo.	**Medio**	**Medio**
Obtener número de columnas	• Obtener el número de columnas	Permitirá obtener el número de columnas establecidas en el grid.	**Bajo**	**Bajo**
Obtener tipo de columna	• Obtener el tipo de columna	Permitirá obtener el valor representativo del tipo de columna.	**Bajo**	**Bajo**
Establecer tipo de columna	• Establecer el tipo de columna.	Permitirá cambiar el tipo de representación de una columna.	**Bajo**	**Bajo**
Obtener el ID único de fila.	• Obtener el ID único de una fila.	Obtiene el ID único asociado a cada fila.	**Bajo**	**Alto**
Establecer la página actual	• Establecer la página actual.	Establece la página actual que se mostrará.	**Medio**	**Medio**
Obtener la página actual	• Obtener la página actual.	Obtiene el número de página que se está mostrando actualmente.	**Bajo**	**Bajo**
Establecer elementos por página	• Establecer elementos por página	Establece el número de elementos que se mostrarán en cada página.	**Medio**	**Medio**
Obtener elementos por página	• Obtener elementos por página.	Obtiene el número de elementos que se mostrarán por cada página.	**Bajo**	**Bajo**
Establecer paginación	• Establecer paginación.	Establece si se debe paginar el grid.	**Bajo**	**Bajo**

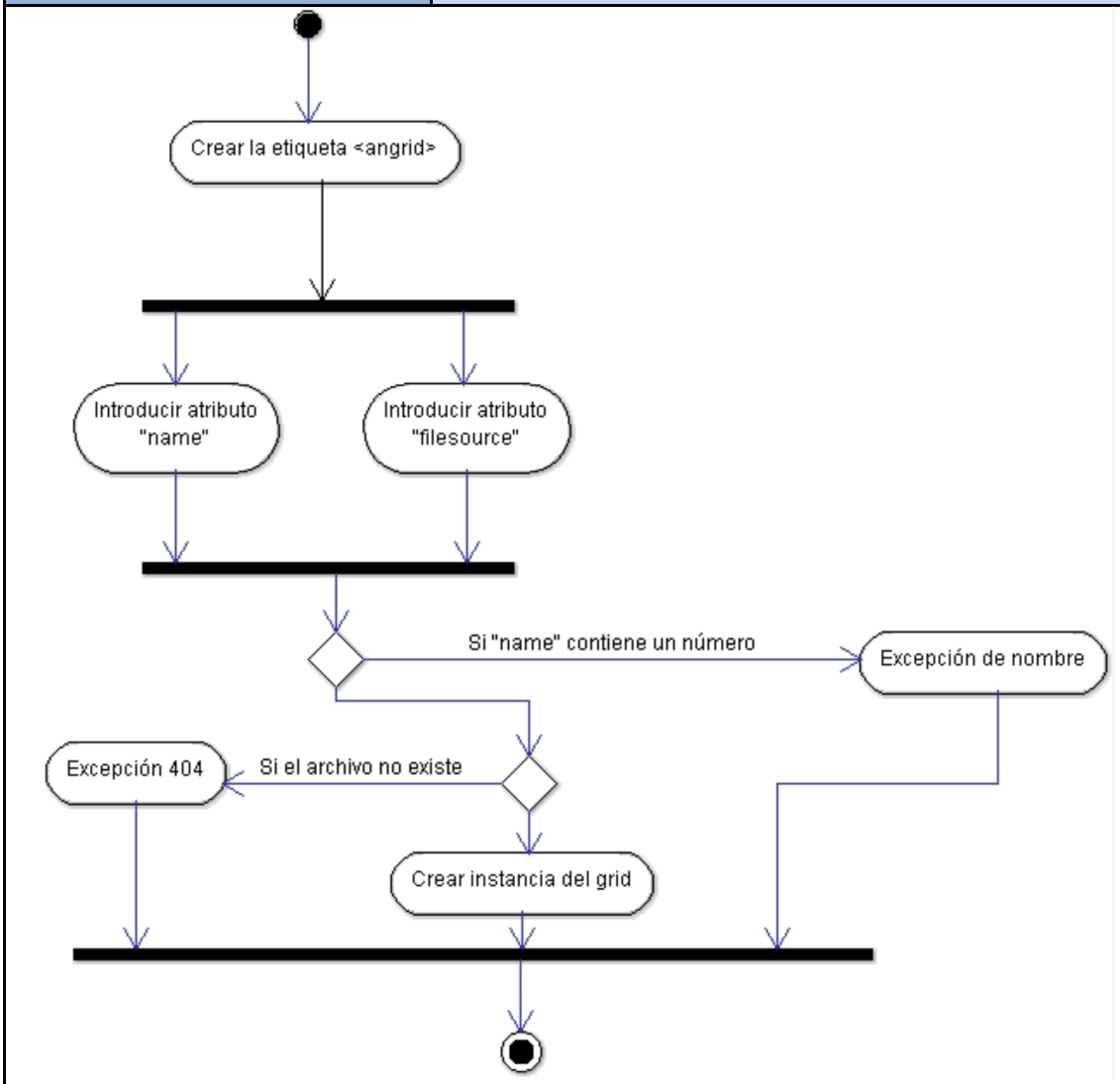
Obtener paginación	• Obtener paginación	Obtiene si se está paginando el grid.	**Bajo**	**Bajo**
Cambiar la fuente de datos	• Cambiar la fuente de datos.	Cambia el modelo que se está mostrando actualmente por otro.	**Alto**	**Alto**

4.6. Modelado de casos de uso

Con el modelado de casos de uso definimos los escenarios y secuencias que se producen para cada caso de uso. En este caso, las secuencias que se definan en negrita estarán realizadas por el grid.

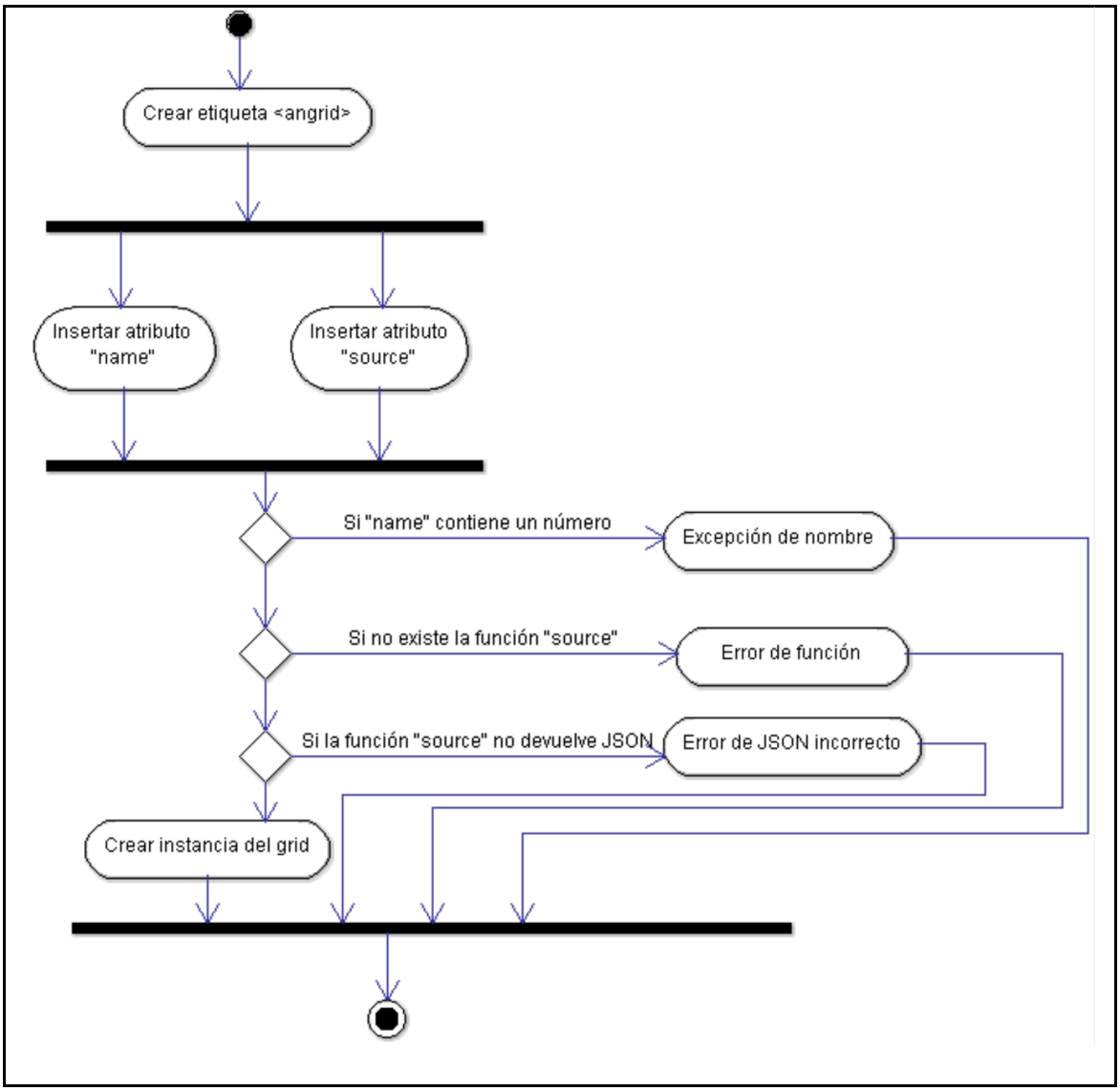
Nombre caso de uso:	**Crear grid desde archivo**
Descripción:	El usuario crea un grid desde un archivo JSON definiendo la aplicación AngularJS y usando la etiqueta ANGRID.
Lista de requisitos cubiertos:	• Formato de entrada. • Entrada desde archivo. • AngularJS. • HTML extendido. • Etiqueta ANGRID. • Definición de la entrada desde archivo. • Requisitos del grid.
Precondiciones:	• El usuario debe haber definido ng-app=”Angrid” en el elemento padre.
Postcondiciones:	• Se creará una nueva instancia de tipo ANGRID.
Escenario principal:	1) El desarrollador introduce la etiqueta ANGRID. 2) Introduce el atributo “name”. 3) Introduce el atributo “filesource” con la ruta al fichero. 4) **Se crea la instancia del grid.**
Escenarios alternativos	• En el paso 2; si no se introduce un nombre o el nombre es un número se producirá una excepción. • En el paso 3; si en filesource se introduce una ruta no válida o hacia un fichero inexistente se producirá una excepción.
Requisitos no funcionales	• Disponibilidad • Portabilidad • Usabilidad

Frecuencia	Una vez cada vez que se desee crear un grid.



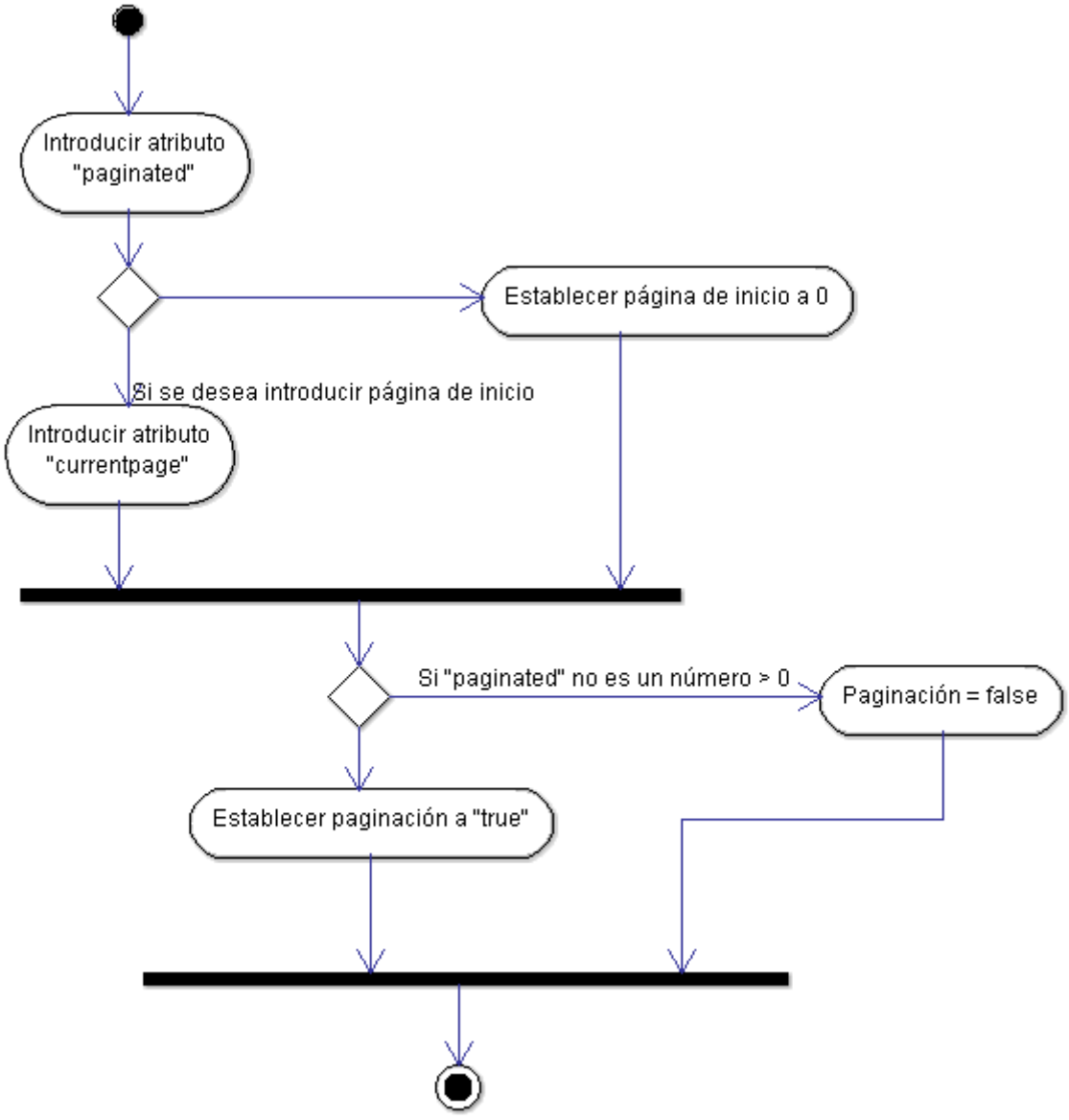
Nombre caso de uso:	**Crear grid desde memoria**
Descripción:	El usuario crea un grid desde un JSON en una función JavaScript definiendo la aplicación AngularJS y usando la etiqueta ANGRID.
Lista de requisitos cubiertos:	• Formato de entrada. • Entrada desde variable. • AngularJS. • HTML extendido. • Etiqueta ANGRID. • Definición de la entrada desde variable. • Requisitos del grid.

Precondiciones:	• El usuario debe haber definido ng-app=”Angrid” en el elemento padre.
Postcondiciones:	• Se creará una nueva instancia de tipo ANGRID.
Escenario principal:	1) El desarrollador introduce la etiqueta ANGRID. 2) Introduce el atributo “name”. 3) Introduce el atributo “source” con el nombre de la función que devuelve el JSON. 4) **Se crea la instancia del grid.**
Escenarios alternativos:	• En el paso 2, si no se introduce un nombre o el nombre es un número se producirá una excepción. • En el paso 3; si la función que contiene “source” no existe se producirá un error al cargar el grid. • En el paso 3; si la función no devuelve un JSON se producirá un error al cargar el grid.
Requisitos no funcionales:	• Disponibilidad • Portabilidad • Usabilidad
Frecuencia:	Una vez cada vez que se desee crear un grid.

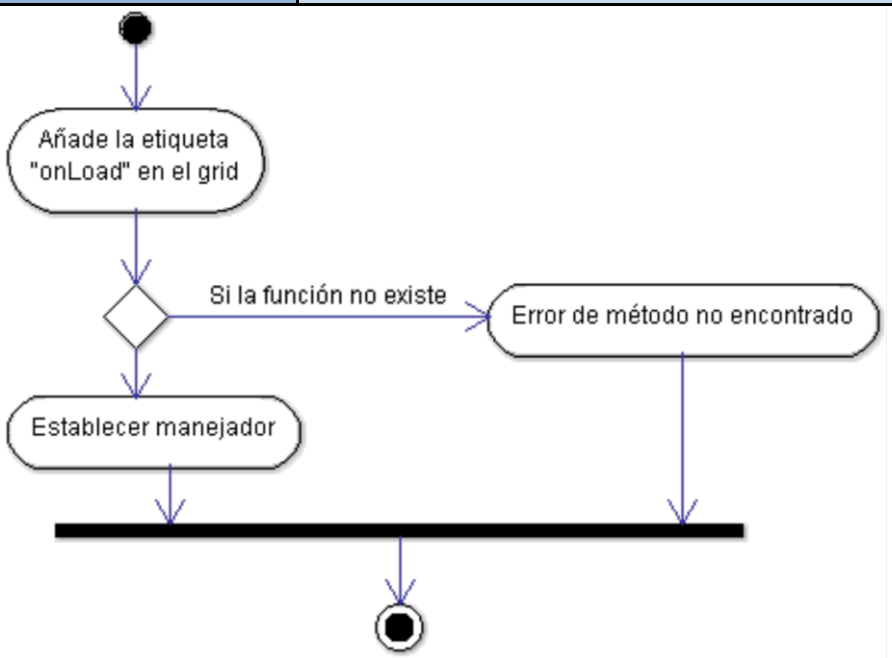


Nombre caso de uso:	**Crear grid paginado**
Descripción:	El usuario crea un grid que pagina los datos de forma automática.
Lista de requisitos cubiertos:	• Paginación. • Elementos por página. • Página actual.
Precondiciones:	• Haber creado un grid desde memoria o fichero.
Postcondiciones:	• El grid dividirá los datos mostrados en páginas.
Escenario principal:	1) Añade la etiqueta “paginated” con el número de elementos en cada página. 2) Añade la etiqueta “currentpage” con la página inicial.

	3) **Establecer valor de "currentpage".** 4) **Establecer el valor de "paginated" a true.**
Escenarios alternativos	• En el paso 1; Si el contenido de "paginated" no es numérico o está vacío no se paginará. • En el paso 1; Si el número es menor que 1 no se paginará. • En el paso 2; Si "currentpage" es nulo o no es numérico la página inicial se establecerá en 0.
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Una vez cada vez que se desee crear un grid.

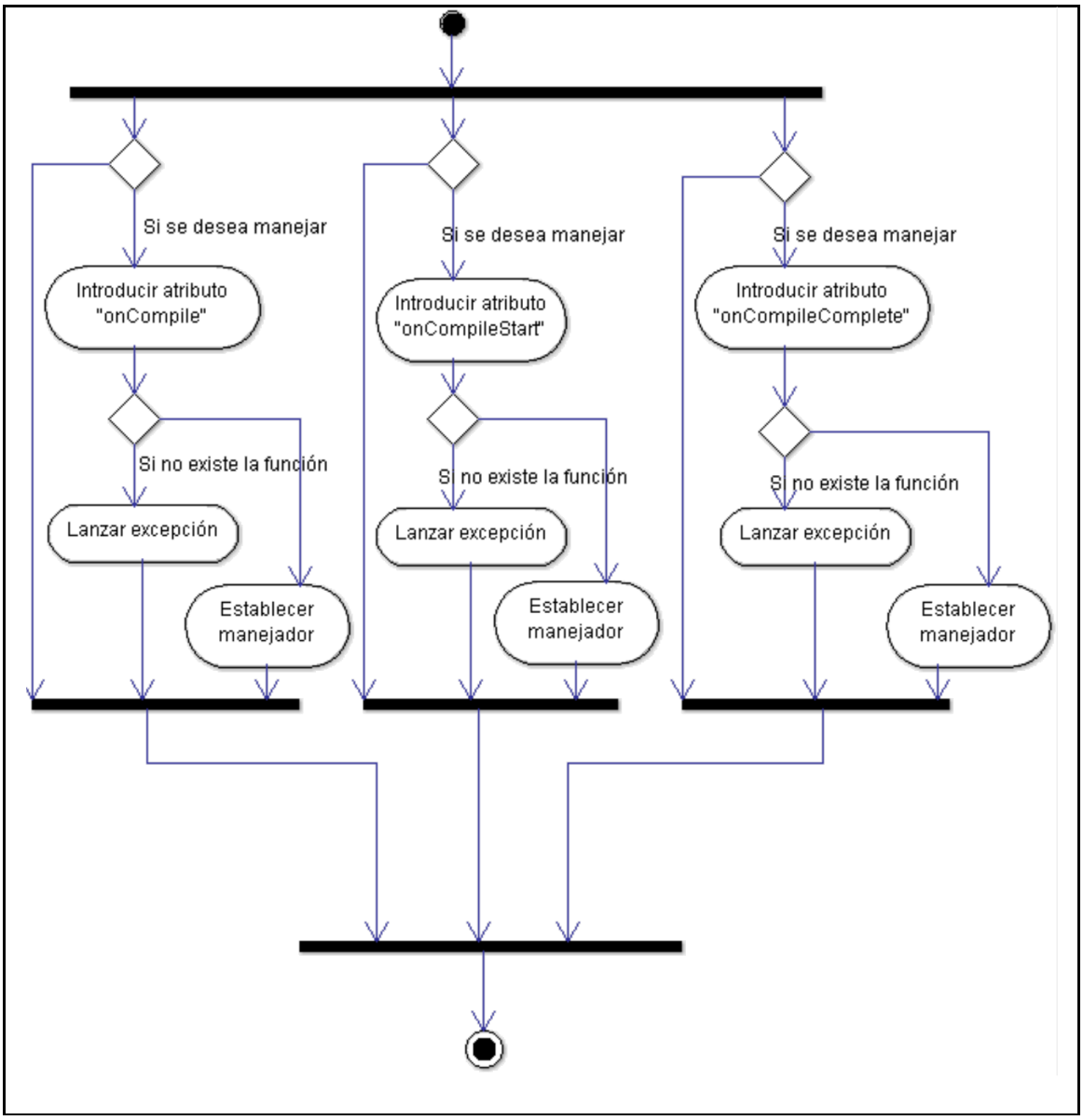


Nombre caso de uso:	**Manejar eventos carga**
Descripción:	El usuario puede realizar funciones al compilar el grid.
Lista de requisitos cubiertos:	• Evento. • Evento onLoad. • Parámetros onLoad.
Precondiciones:	• Crear un grid desde archivo o memoria.
Postcondiciones:	• Se ejecutará la función introducirá al compilar el grid.
Escenario principal:	1) Añade el atributo "onLoad" en el grid con el nombre de la función que se ejecutará. 2) **Establecer manejador en el grid.**
Escenarios alternativos	• En el paso 1; Si no existe ninguna función con ese nombre se producirá un error.
Requisitos no funcionales:	• Interoperabilidad • Usabilidad • Mantenibilidad
Frecuencia:	Cada vez que cargue la página con el grid.



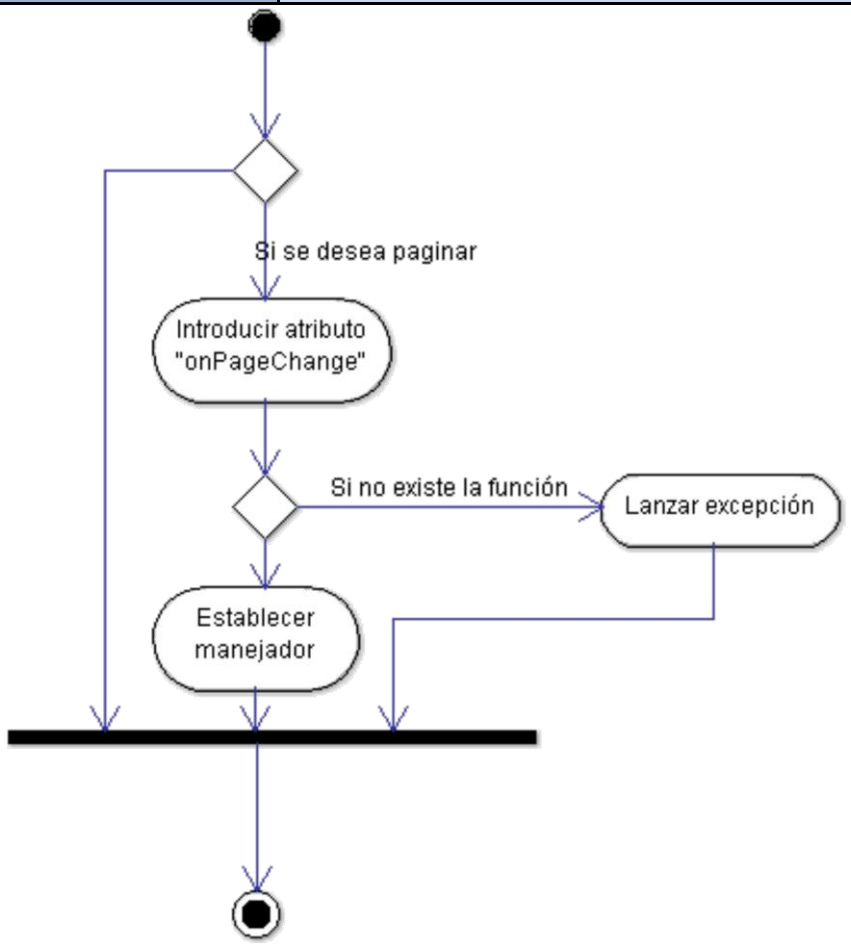
Nombre caso de uso:	**Manejar eventos de compilación**

Descripción:	El usuario puede realizar funciones al compilar el grid.
Lista de requisitos cubiertos:	• Evento. • Evento onCompileStart. • Evento onCompile. • Evento onCompileComplete. • Parámetros onCompileStart. • Parámetros onCompile. • Parámetros onCompileStart. • Fachada.
Precondiciones:	• Debe haberse creado un grid desde memoria o desde archivo.
Postcondiciones:	• Se ejecutarán las funciones que se hayan introducido.
Escenario principal:	1) Introduce el atributo onCompileStart al grid con el nombre de la función que se ejecutará. 2) **Establecer manejador onCompileStart.** 3) Introduce el atributo onCompile al grid con el nombre de la función que se ejecutará. 4) **Establecer manejador onCompile.** 5) Introduce el atributo onCompileComplete al grid con el nombre de la función que se ejecutará. 6) **Establecer manejador onCompileComplete.**
Escenarios alternativos:	• En todos los pasos; Si no se introduce el atributo se omitirá el evento. • En todos los pasos; Si no existe ninguna función con el nombre introducido se producirá un error.
Frecuencia:	• Cada vez que se recompile el DOM del grid.
Requisitos no funcionales:	• Interoperabilidad • Usabilidad • Mantenibilidad



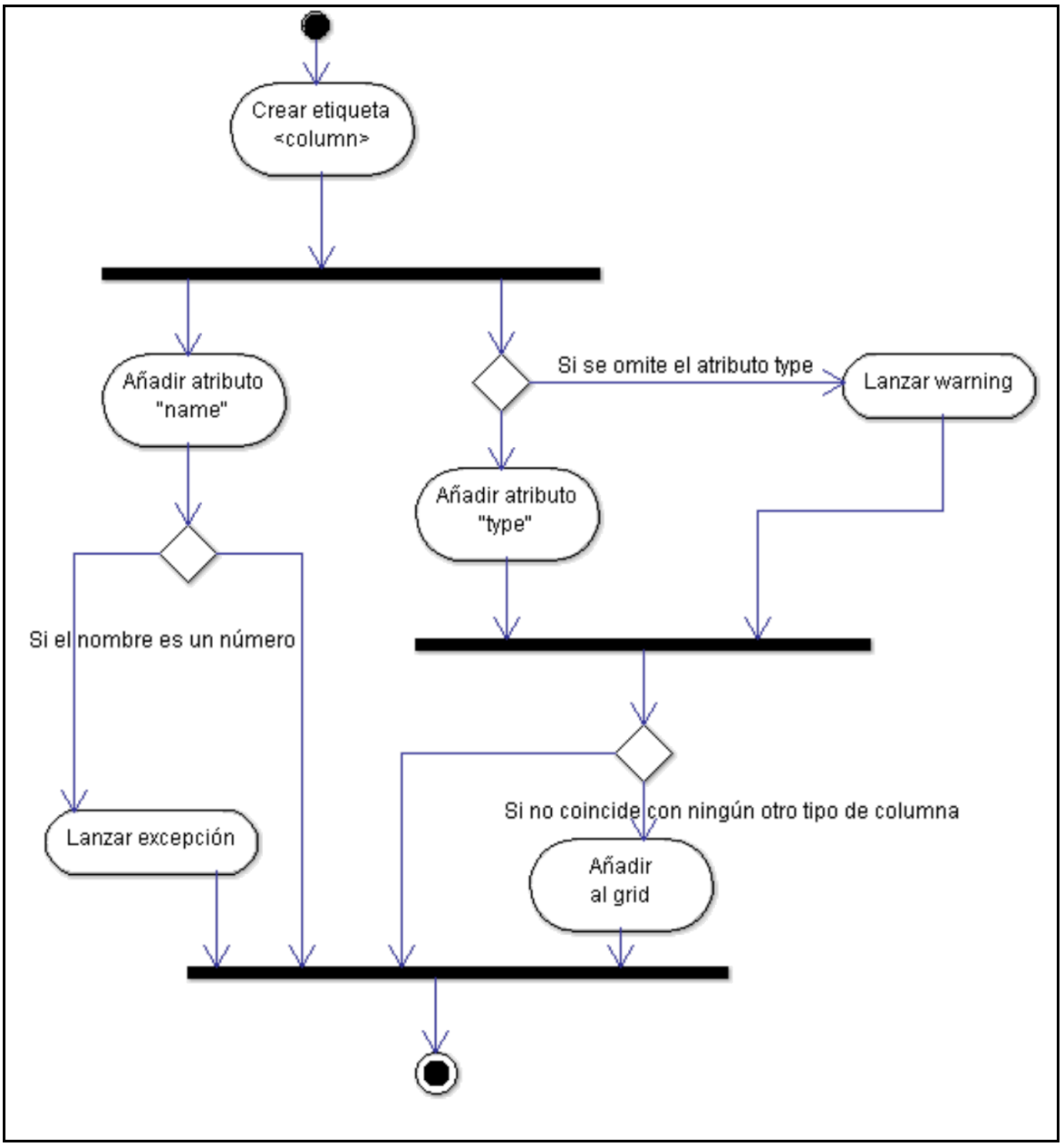
Nombre caso de uso:	**Manejar eventos de paginado**
Descripción:	El usuario puede realizar funciones cuando se cambia de página en el grid.
Lista de requisitos cubiertos:	• Evento. • Evento onPageChange. • Parámetros onPageChange.
Precondiciones:	• Debe haberse creado un grid desde archivo o desde memoria.
Postcondiciones	• Se ejecutará la función determinada por el evento.

Escenario principal:	1) Introduce el atributo “onPageChange” dentro del grid con el nombre de la función que se ejecutará. 2) **Establecer manejador onPageChange.**
Escenarios alternativos:	• En el paso 1; Si no se introduce el atributo el evento se omitirá. • En el paso 1; Si no existe ninguna función con el nombre introducido se producirá una excepción.
Requisitos no funcionales:	• Interoperabilidad • Usabilidad • Mantenibilidad
Frecuencia:	Cada vez que cambie la página actual del grid.



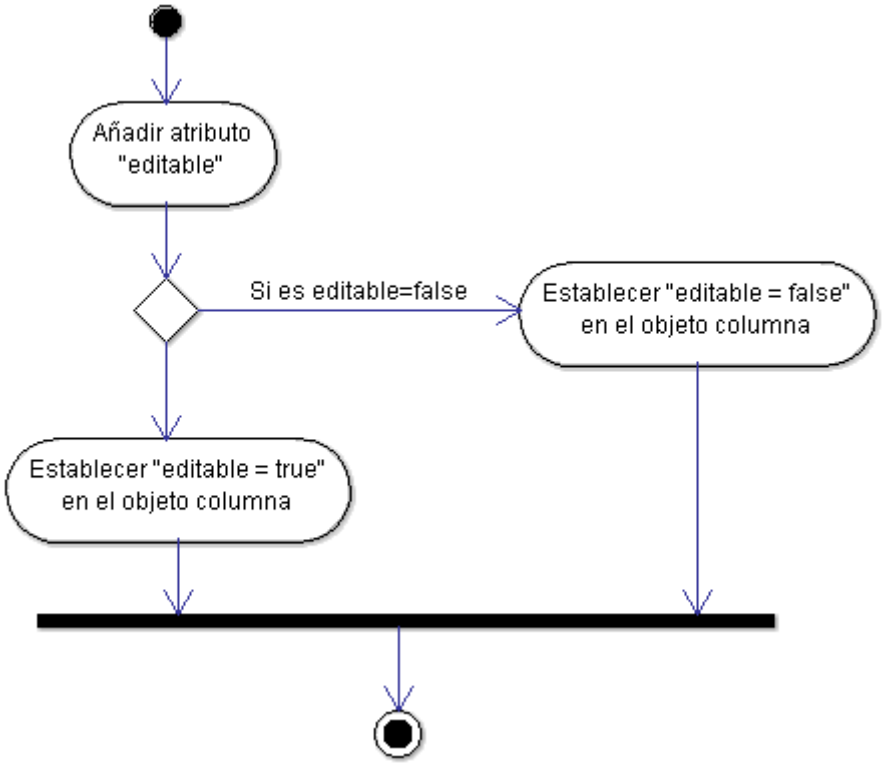
Nombre caso de uso:	**Crear columna de tipo texto**
Descripción:	El usuario crea una columna cuyo contenido será un

	texto.
Lista de requisitos cubiertos:	• HTML extendido. • Etiqueta COLUMN. • Posición COLUMN. • Nombre de la columna. • Nombre de columna válido. • Nombre para mostrar. • Tipos de columnas. • Columnas texto.
Precondiciones:	• Debe tener como padre una etiqueta <angrid>.
Postcondiciones:	• Se introducirá la columna en la lista de columnas del grid como tipo columna texto no editable.
Escenario principal:	1) Crear etiqueta <column>. 2) Introducir el atributo “name”. 3) Introducir el atributo “type=’text’”. 4) **Añadir columna al grid.**
Escenarios alternativos:	• En el paso 2; Si el atributo “name” contiene un número se lanzará una excepción. • En el paso 3; Si se omite el atributo se preestablecerá de tipo texto. • En el paso 3; Si el atributo “type” contiene un tipo de columna inexistente se preestablecerá de tipo texto.
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Por cada columna de tipo texto que se quiera añadir al grid.



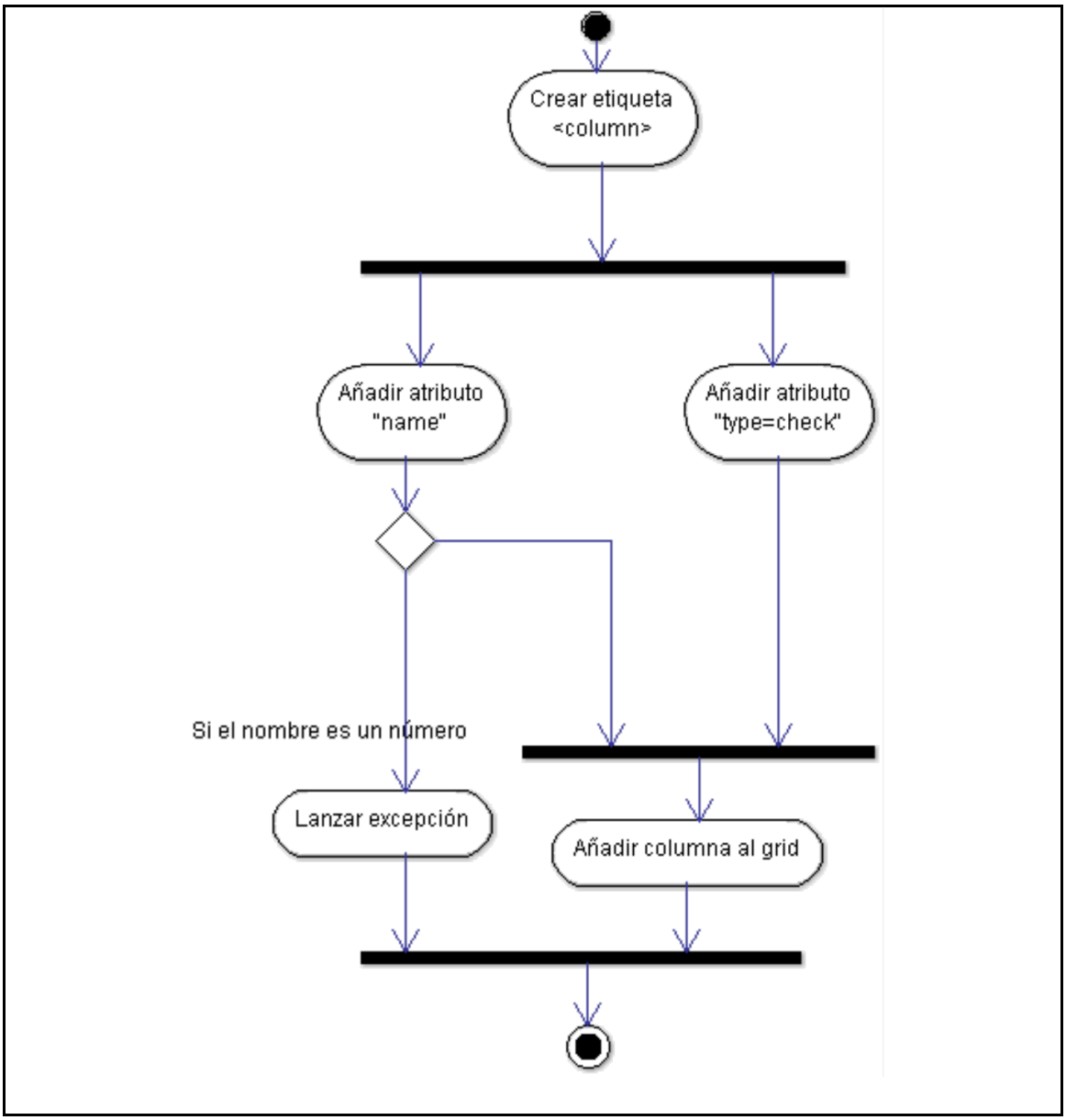
Nombre caso de uso:	**Crear columna de texto editable**
Descripción:	El usuario crea una columna cuyo contenido será un texto que se podrá modificar en tiempo de ejecución.
Lista de requisitos cubiertos:	• Columnas editables. • Textos editables.
Precondiciones:	• Crear una columna de texto.
Postcondiciones:	• Se podrá editar la columna en tiempo de ejecución.

Escenario principal:	1) Añadir atributo “editable” a la columna.
Escenarios alternativos:	• En el paso 1; Si se introduce “editable = false” no será editable.
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Cada vez que se quiera hacer editable una columna de tipo texto.



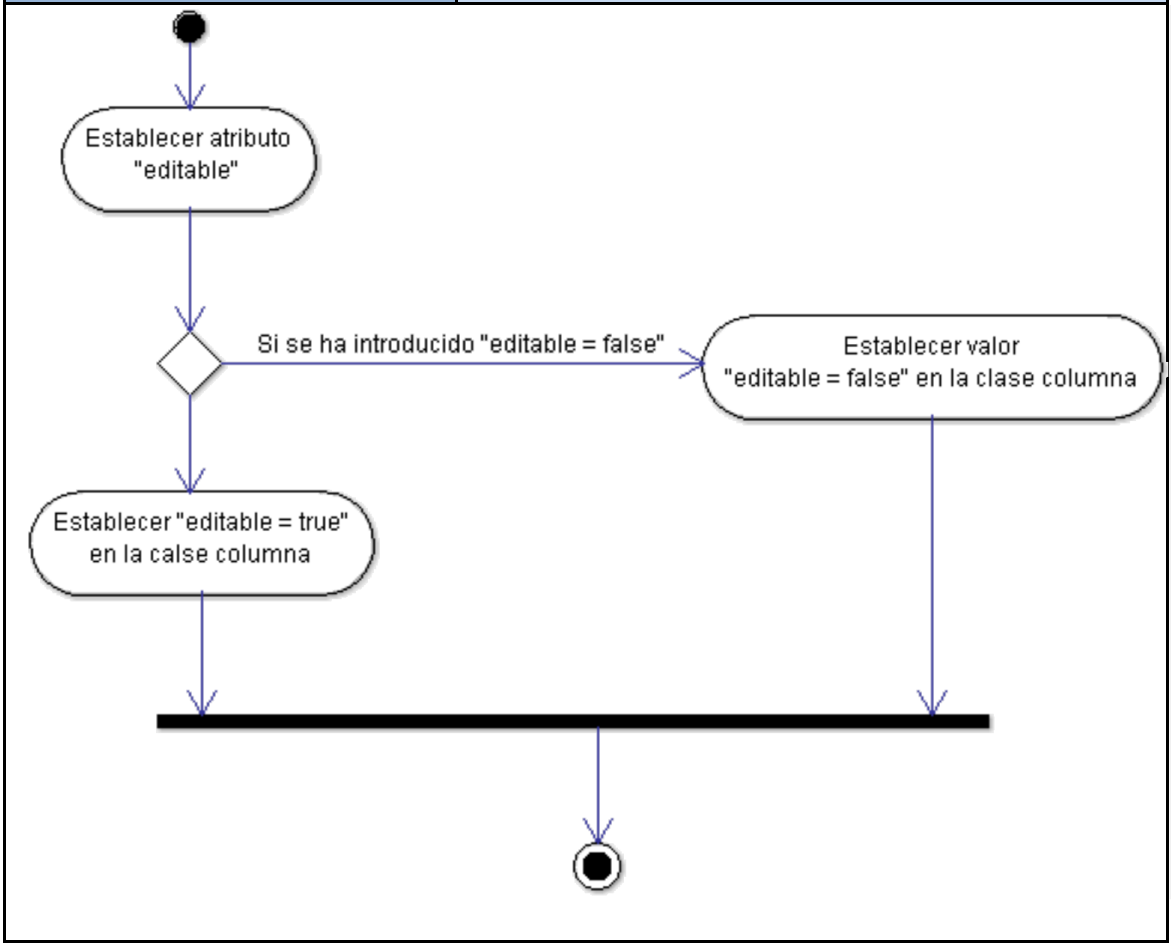
Nombre caso de uso:	**Crear columna con un checkbox**
Descripción:	El usuario crea una columna de tipo *check* cuyo contenido será un checkbox.
Lista de requisitos cubiertos:	• HTML extendido. • Etiqueta COLUMN. • Posición COLUMN. • Nombre de la columna. • Nombre de columna válido. • Nombre para mostrar. • Tipos de columnas. • Columnas checkbox.

Precondiciones:	• Debe tener como padre la etiqueta <angrid>.
Postcondiciones:	• Se introducirá la columna en la lista de columnas del grid como tipo columna check no editable.
Escenario principal:	1) Crear etiqueta <column>. 2) Introducir el atributo “name”. 3) Introducir el atributo “type=’check’”. 4) **Añadir columna al grid.**
Escenarios alternativos:	• En el paso 2; Si el atributo “name” contiene un número se lanzará una excepción.
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Cada vez que se quiera añadir una columna de con checkboxes.



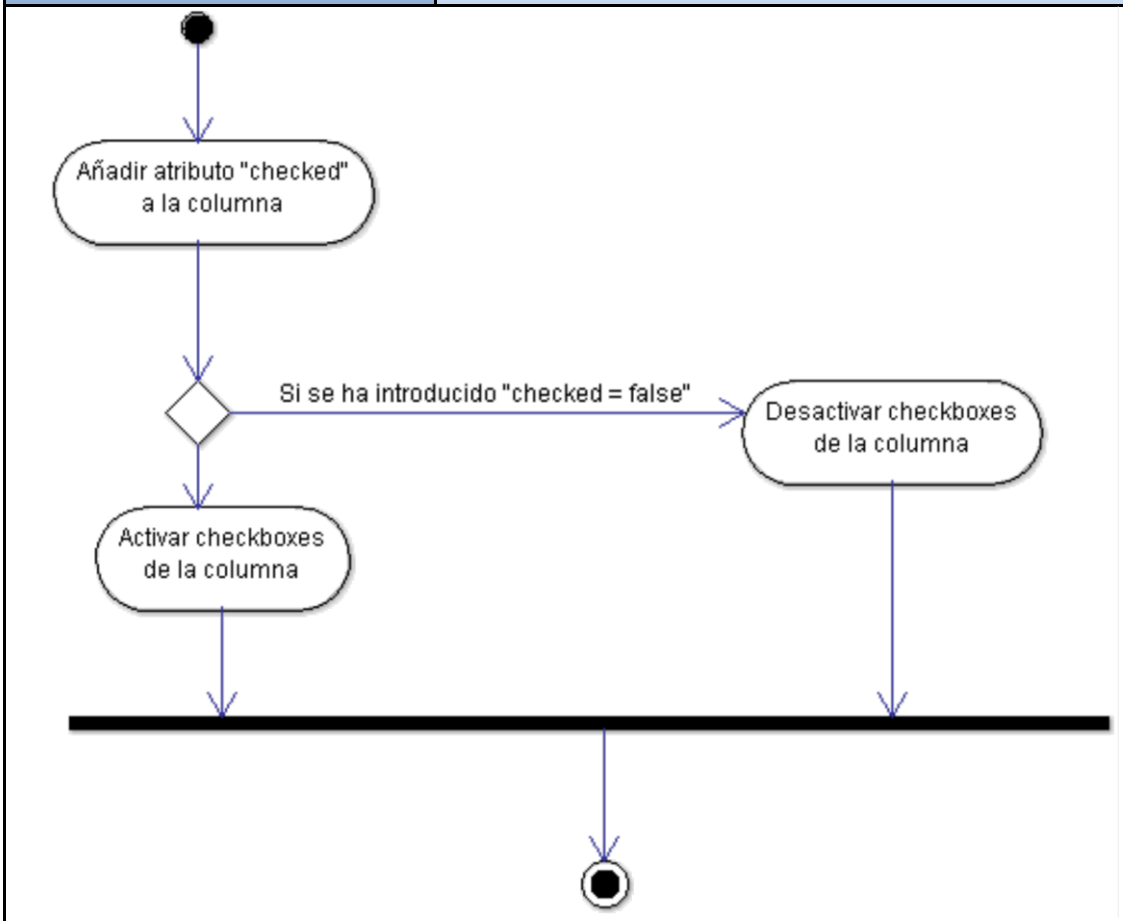
Nombre caso de uso:	**Crear columna con checkbox cambiable**
Descripción:	El usuario crea una columna de tipo *check* con un checkbox que cambia su estado al hacerle click.
Lista de requisitos cubiertos:	• Columnas editables. • Checkbox editables.
Precondiciones:	• Tener una columna de tipo checkbox.
Postcondiciones:	• El contenido de la columna podrá variarse haciendo click.

Escenario principal:	1) Añadir el atributo “editable” en la columna.
Escenarios alternativos:	• En el paso 1; Si se introduce “editable = false” no se podrá cambiar el checkbox.
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Cada vez que se desee poner un checkbox que cambie al hacer click en él.



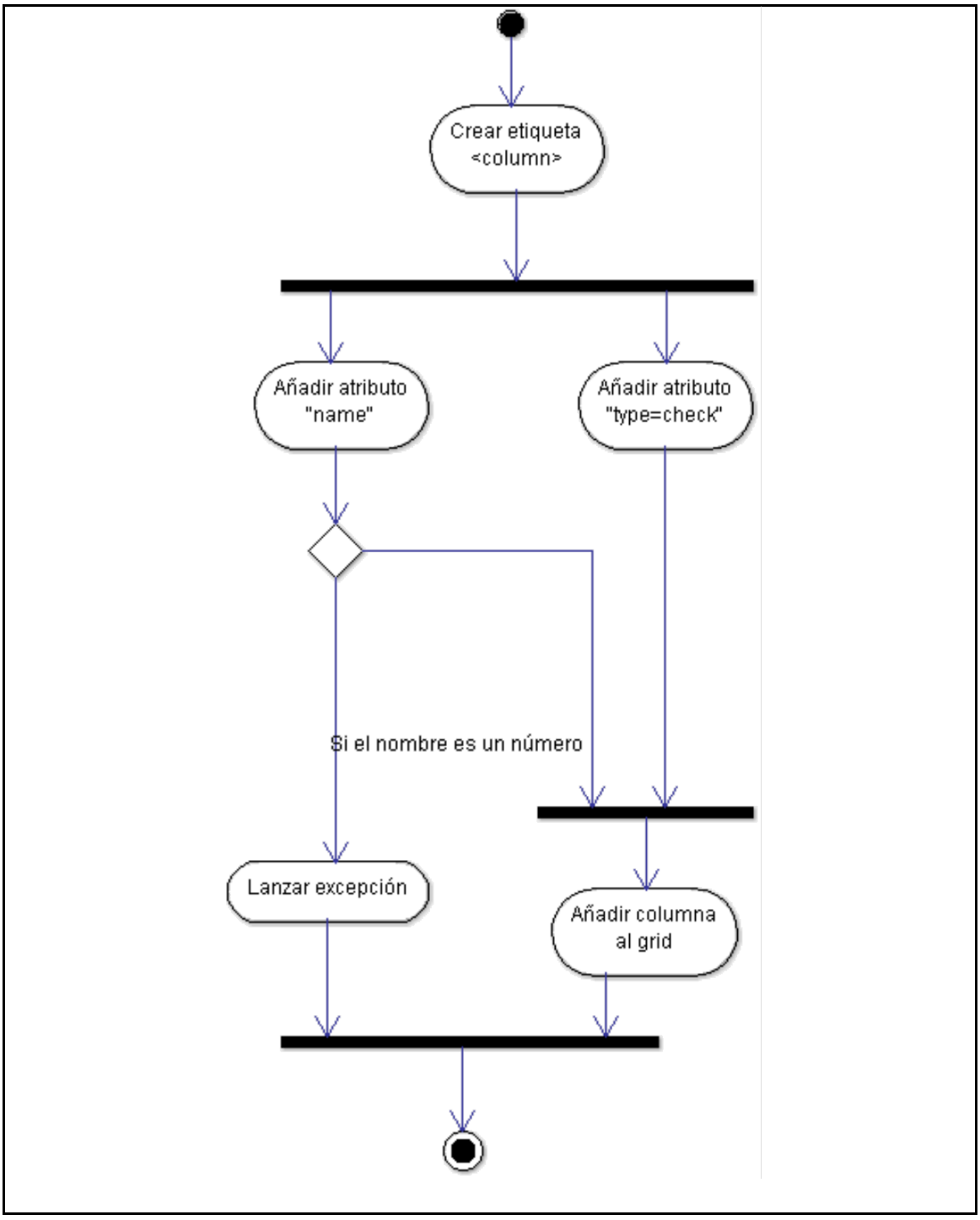
Nombre caso de uso:	**Checkboxes ya activados en las columnas**
Descripción:	El usuario puede establecer el valor del checkbox sin *source*.
Lista de requisitos cubiertos:	• Checkboxes activados.
Precondiciones:	• Se debe haber creado una columna de tipo checkbox.
Postcondiciones:	• El checkbox de las columnas estará activado.

Escenario principal:	1) Añadir atributo “checked” a la columna.
Escenarios alternativos:	• En el paso 1; Si se introduce “checked = false” el checkbox no será activado.
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Cada vez que se quiera introducir una columna con checkboxes y manejar manualmente su activación al inicio.



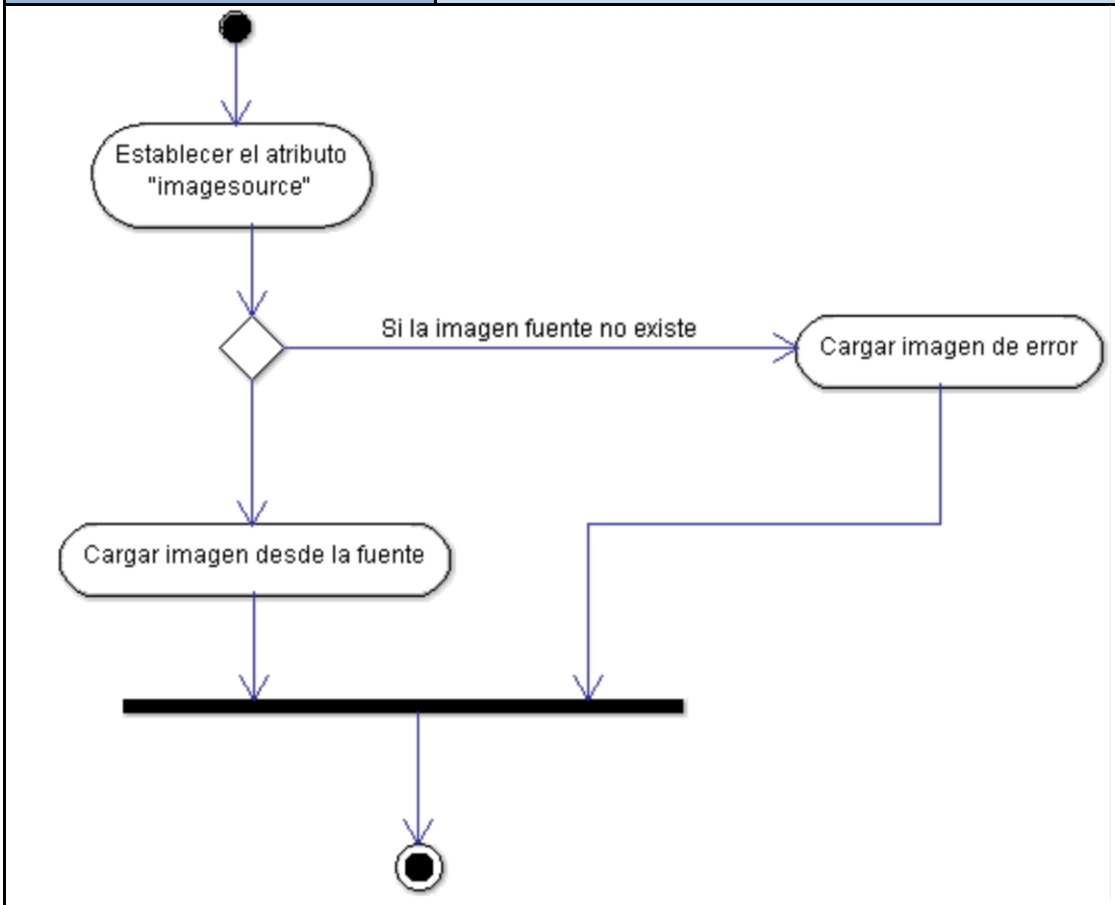
Nombre caso de uso:	**Crear columnas con imágenes**
Descripción:	El usuario crea una columna cuyo contenido será la URL de una imagen que se mostrará.
Lista de requisitos cubiertos:	• HTML extendido. • Etiqueta COLUMN. • Posición COLUMN. • Nombre de la columna. • Nombre de columna válido. • Nombre para mostrar.

	• Tipos de columnas. • Columnas imagen.
Precondiciones:	• Debe tener como padre la etiqueta <angrid>.
Postcondiciones:	• Se introducirá la columna en la lista de columnas del grid como tipo columna imagen no editable.
Escenario principal:	1) Crear etiqueta <column>. 2) Introducir el atributo "name". 3) Introducir el atributo "type='image'". 4) **Añadir columna al grid.**
Escenarios alternativos:	• En el paso 2; Si el atributo "name" contiene un número se lanzará una excepción.
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Cada vez que se quiera añadir una columna de con imágenes.



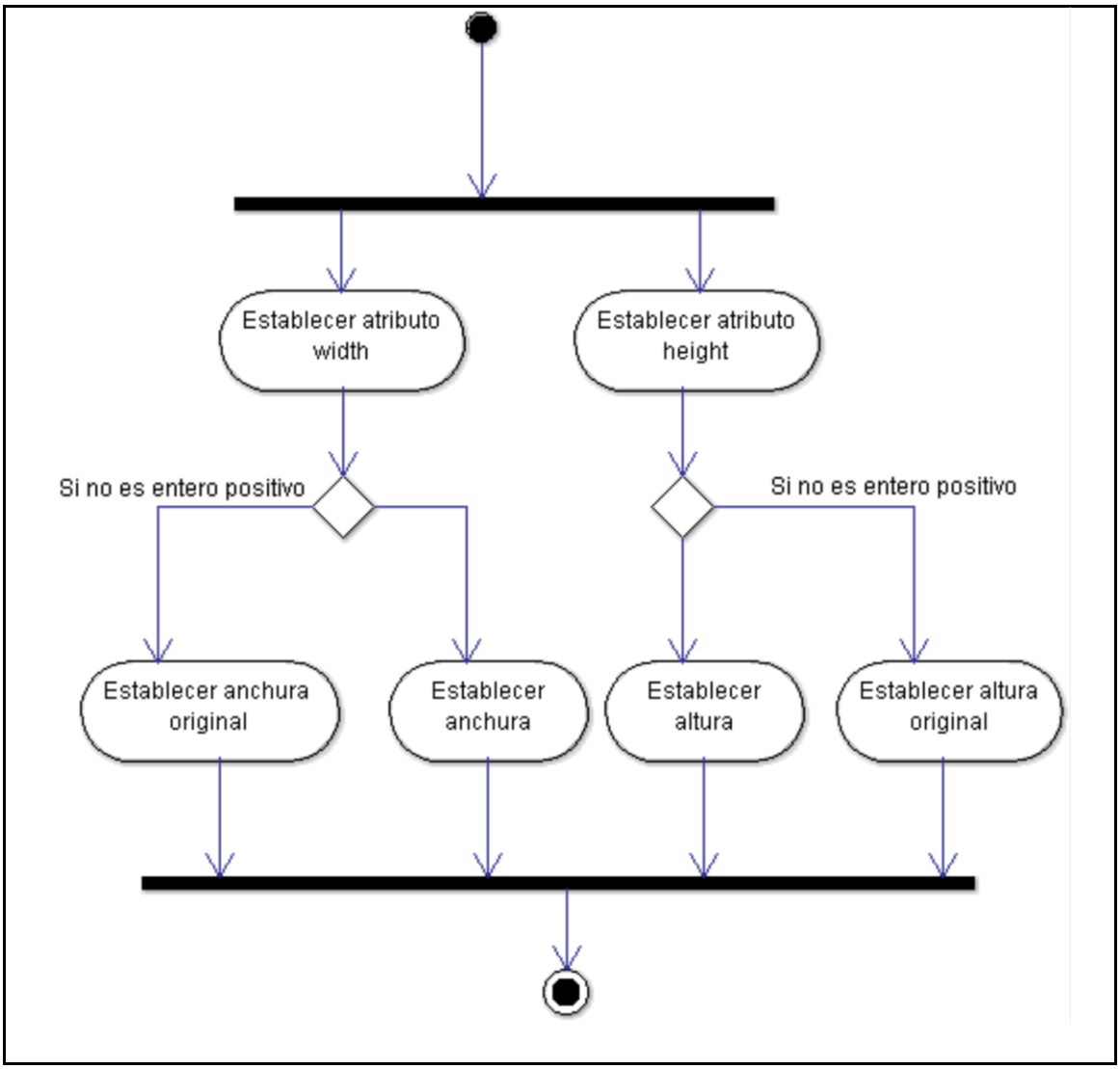
Nombre caso de uso:	**Cargar imágenes en una columna**
Descripción:	El usuario crea una columna cuya imagen no está determinada por los datos de entrada.
Lista de requisitos cubiertos:	• Imagen predeterminada.

Precondiciones:	• Tener una columna de tipo imagen.
Postcondiciones:	• La imagen cargada será la que se introduzca en el atributo.
Escenario principal:	1) Añadir el atributo “imagesource”. 2) **Cargar fuente de imagen en la columna.**
Escenarios alternativos:	• En el paso 1; Si la fuente de la imagen es errónea se cargará la imagen de error en su lugar.
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Cada vez que el desarrollador quiera establecer manualmente la imagen de una columna.



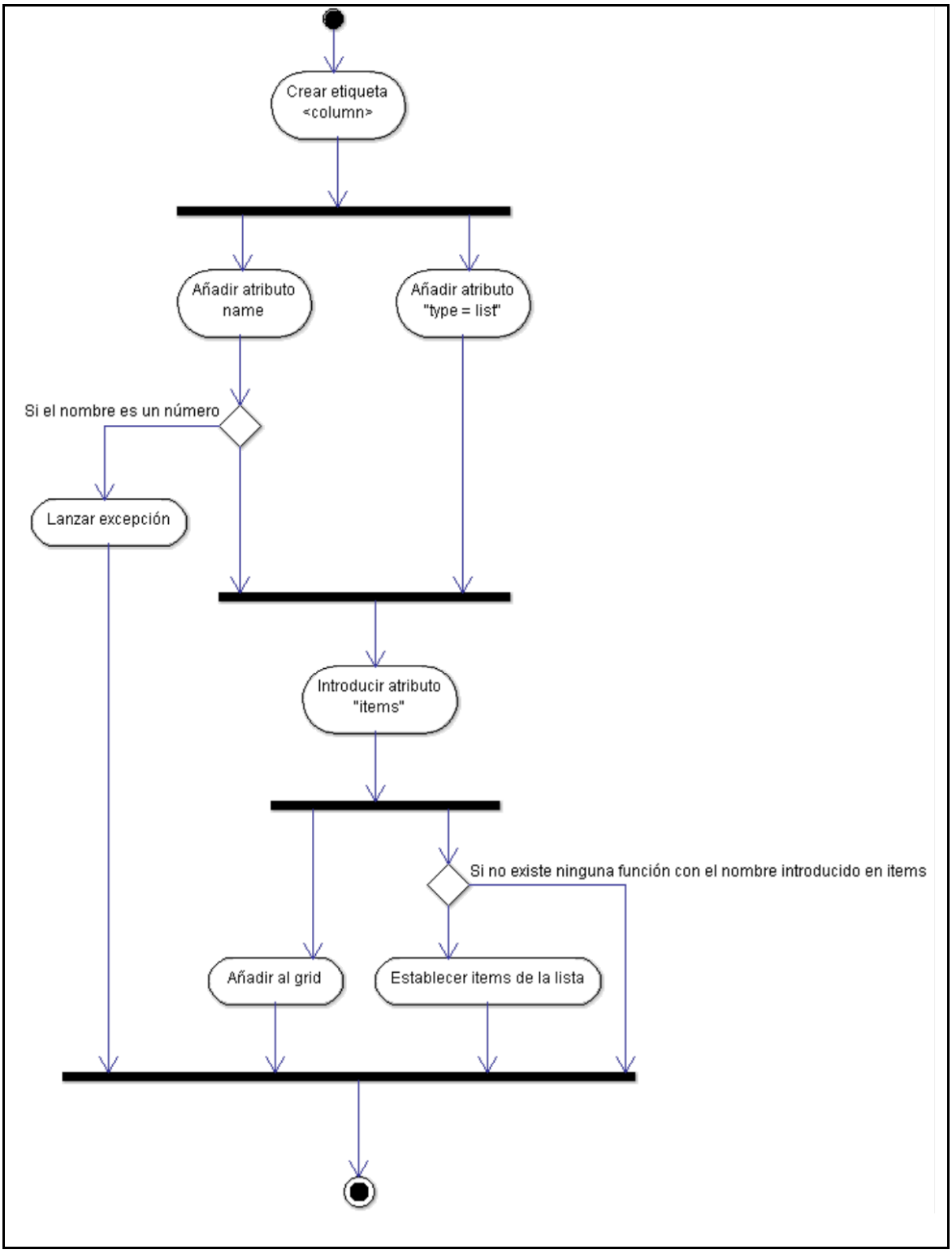
Nombre caso de uso:	**Establecer el tamaño de las imágenes**
Descripción:	El usuario establece manualmente el tamaño de las imágenes.

Lista de requisitos cubiertos:	• Tamaño imágenes.
Precondiciones:	• Tener una columna de tipo imagen.
Postcondiciones:	• El tamaño de la imagen vendrá determinado por los atributos introducidos manualmente.
Escenario principal:	1) Añadir el atributo “width”. 2) Añadir el atributo “height”. 3) **Establecer el atributo “witdh” en la columna.** 4) **Establecer el atributo “height” en la columna.**
Escenarios alternativos:	• En el paso 3 y 4; Si el valor es negativo o no numérico se establecerá el tamaño predeterminado de la imagen.
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Cada vez que el desarrollador quiera modificar el tamaño de las imágenes que se cargan en las columnas.



Nombre caso de uso:	**Crear columnas con listas**
Descripción:	El usuario crea una columna cuyo control sea la selección de un elemento de una lista.
Lista de requisitos cubiertos:	• HTML extendido. • Etiqueta COLUMN. • Posición COLUMN. • Nombre de la columna. • Nombre de columna válido. • Nombre para mostrar. • Tipos de columnas. • Columnas lista. • Items de la lista.
Precondiciones:	• Debe tener como padre la etiqueta <angrid>.

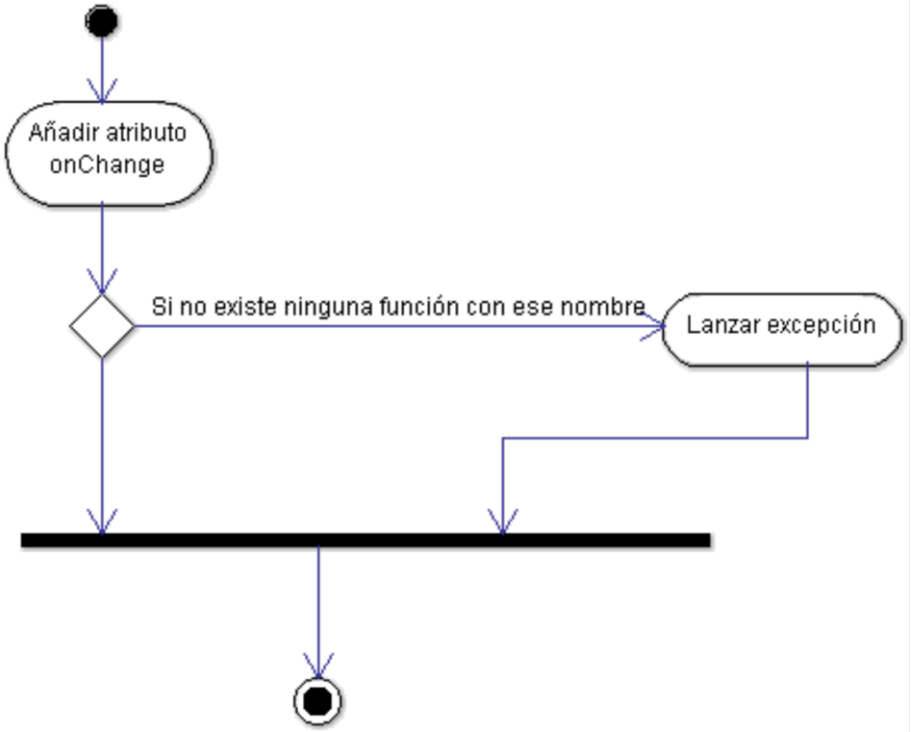
Postcondiciones:	• Se introducirá la columna en la lista de columnas del grid como tipo columna lista editable.
Escenario principal:	1) Crear etiqueta <column>. 2) Introducir el atributo “name”. 3) Introducir el atributo “type=’list’”. 4) Introducir el atributo “editable”. 5) Introducir el atributo “ítems” con la función que devolverá el JSON. 6) **Añadir columna al grid.** 7) **Establecer ítems de la lista.**
Escenarios alternativos:	• En el paso 2; Si el atributo “name” contiene un número se lanzará una excepción. • En el paso 5; Si no existe ninguna función con el nombre introducido se producirá una excepción.
Requisitos no funcionales:	• Portabilidad • Usabilidad • Interoperabilidad • Mantenibilidad
Frecuencia:	Cada vez que se quiera añadir una columna de con listas seleccionables.



Nombre caso de uso:	**Columnas con listas desactivadas**
Descripción:	El usuario desactiva la selección de la lista usando el parámetro *editable.*

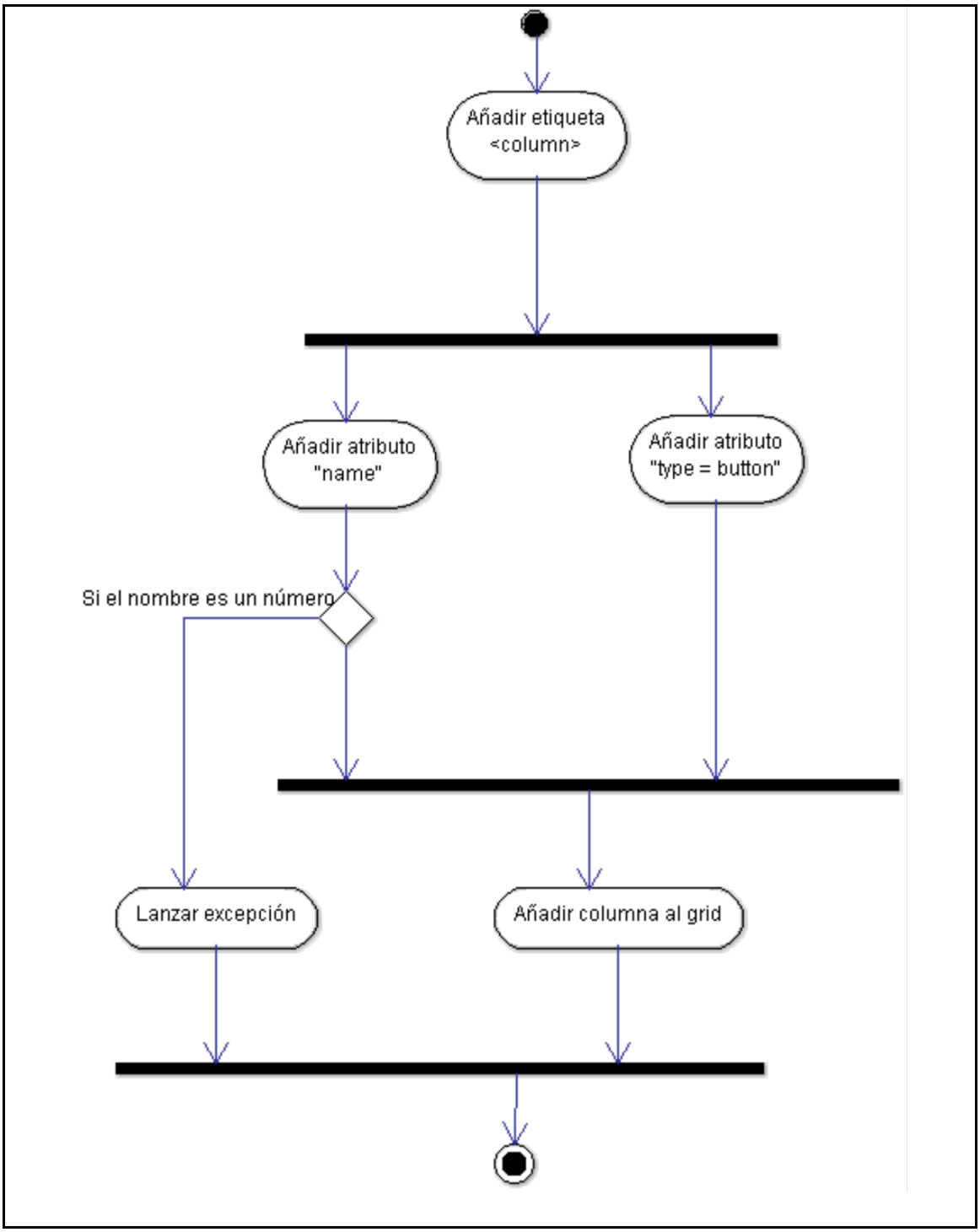
Lista de requisitos cubiertos:	• Listas editables.
Precondiciones:	• Tener una columna de tipo lista.
Postcondiciones:	• No se podrán seleccionar elementos en la lista.
Escenario principal:	1) Añadir atributo “editable = true”.
Escenarios alternativos:	
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Cada vez que el desarrollador quiera desactivar la selección en las columnas.
	

Nombre caso de uso:	**Manejar el cambio de selección de la lista**
Descripción:	Usar el evento onChange para manejar cuándo cambie la selección de la lista de una celda.
Lista de requisitos cubiertos:	• Evento. • OnChange. • Parámetros OnChange.
Precondiciones:	• Tener una columna de tipo lista. • La lista no puede estar desactivada.
Postcondiciones:	• Se lanzará la función al cambiar la selección de la lista.
Escenario principal:	1) Añadir el atributo “onChange” en la columna con el nombre de la función. 2) **Añadir manejador onChange a la columna.**
Escenarios alternativos:	• En el paso 1; Si no existe ninguna función con

	ese nombre se producirá una excepción.
Requisitos no funcionales:	• Interoperabilidad • Usabilidad • Mantenibilidad
Frecuencia:	Cada vez que se quiera ejecutar una función cuando se cambie la selección de la lista.
	

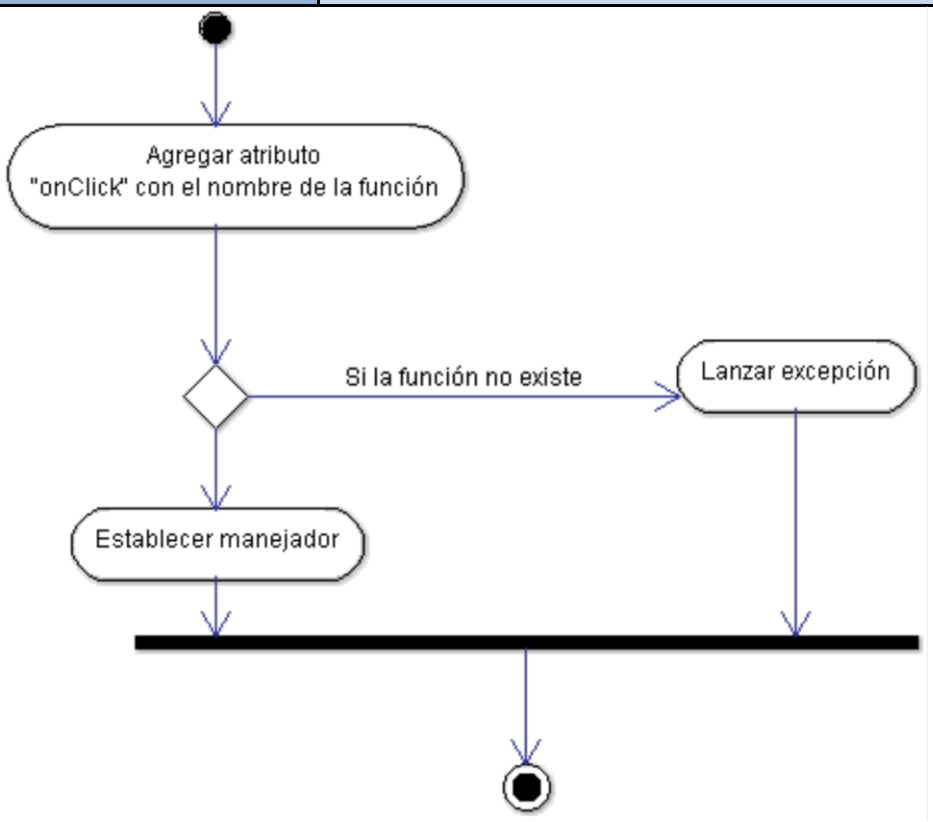
Nombre caso de uso:	**Crear columnas con botones**
Descripción:	El usuario crea una columna cuyo contenido serán botones con el texto del modelo de datos.
Lista de requisitos cubiertos:	• HTML extendido. • Etiqueta COLUMN. • Posición COLUMN. • Nombre de la columna. • Nombre de columna válido. • Nombre para mostrar. • Tipos de columnas. • Columna botón.
Precondiciones:	• Debe tener como padre la etiqueta <angrid>.

Postcondiciones:	• Se introducirá la columna en la lista de columnas del grid como tipo columna de botones.
Escenario principal:	5) Crear etiqueta <column>. 6) Introducir el atributo “name”. 7) Introducir el atributo “type=’button’”. 8) **Añadir columna al grid.**
Escenarios alternativos:	• En el paso 2; Si el atributo “name” contiene un número se lanzará una excepción.
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Cada vez que se quiera añadir una columna con botones.



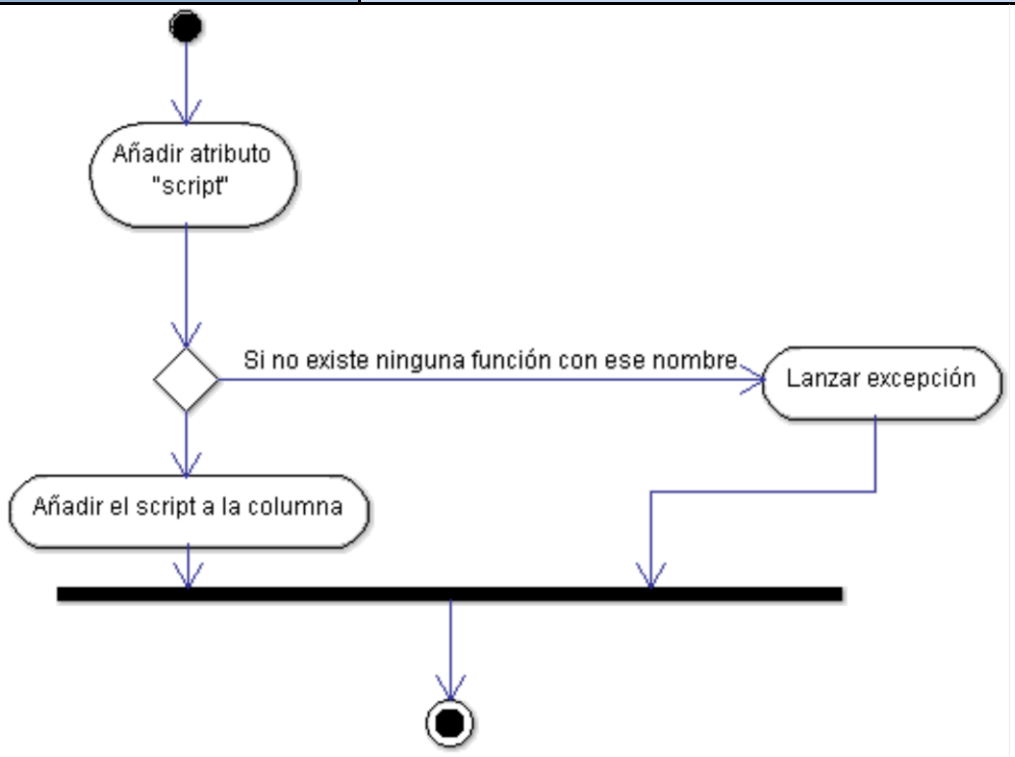
Nombre caso de uso:	**Manejar el click en las columnas**
Descripción:	El usuario puede definir la función que manejará el evento de hacer click sobre una celda de una columna.
Lista de requisitos cubiertos:	• Evento. • OnClick.

	• Parámetros OnClick.
Precondiciones:	• Tener una columna de cualquier tipo.
Postcondiciones:	• Se ejecutará la función JavaScript cuando el usuario final haga click sobre la función.
Escenario principal:	1) Añadir atributo “onClick”. 2) **Añadir manejador onClick de la columna.**
Escenarios alternativos:	• En el paso 1; Si no existe ninguna función con el nombre introducido se lanzará una excepción.
Requisitos no funcionales:	• Interoperabilidad • Usabilidad • Mantenibilidad
Frecuencia:	Cada vez que se quiera manejar cuándo el usuario final hace click sobre una celda.



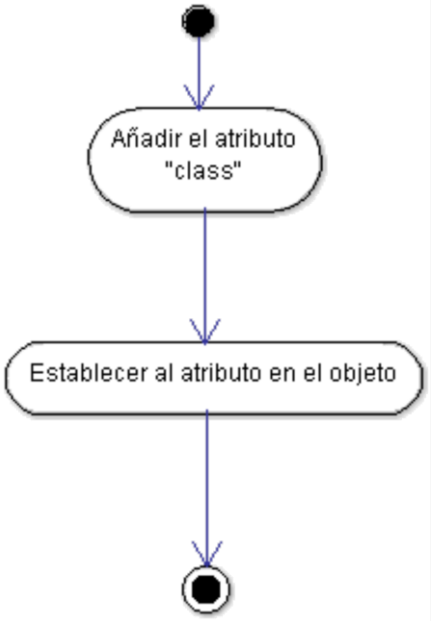
Nombre caso de uso:	**Columnas con script**
Descripción:	El usuario puede definir una función que se ejecutará

	para todas las filas de una columna.
Lista de requisitos cubiertos:	• Fachada.
Precondiciones:	• Tener una columna.
Postcondiciones:	• Se ejecutará la función al cargar la columna.
Escenario principal:	1) Añadir el atributo "script" en la columna. 2) **Establecer el valor del script en la columna.**
Escenarios alternativos:	• En el paso 1; Si no existe ninguna función con el nombre introducido se lanzará una excepción.
Requisitos no funcionales:	• Interoperabilidad • Usabilidad • Mantenibilidad
Frecuencia:	Cada vez que se quiera ejecutar una función para una columna al cargar la vista.



Nombre caso de uso:	**Formato del grid**
Descripción:	El usuario puede añadir hojas de estilos y asignarles clases de estilos CSS a los elementos del grid utilizando el

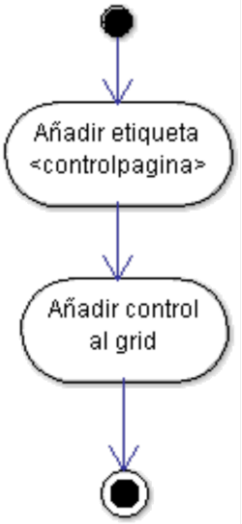
	atributo *class.*
Lista de requisitos cubiertos:	• Estilo del grid.
Precondiciones:	• Debe existir un elemento del grid.
Postcondiciones:	• El elemento con el atributo tendrá el estilo CSS marcado.
Escenario principal:	1) Añadir el atributo "class" con el nombre de la clase CSS. 2) **Establecer clase en el elemento.**
Escenarios alternativos:	
Requisitos no funcionales:	• Usabilidad • Interoperabilidad • Mantenibilidad
Frecuencia:	Cada vez que se le quiera dar un formato a un elemento del grid.



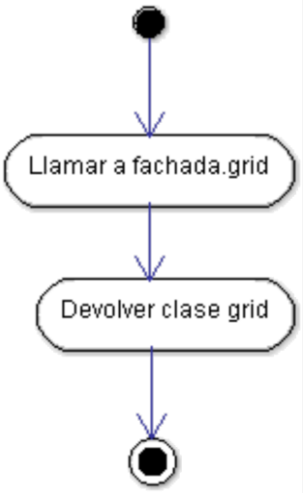
Nombre caso de uso:	**Ordenación del grid**
Descripción:	Las columnas que tengan el atributo *sortable* podrán ser clave para la ordenación de los datos.
Lista de requisitos cubiertos:	• Columnas ordenables.

Precondiciones:	• Tener una columna.
Postcondiciones:	• La columna se podrá ordenar haciendo click sobre la cabecera.
Escenario principal:	1) Añadir el atributo “sortable” a la columna. 2) **Establecer valor de sortable en la columna.**
Escenarios alternativos:	• En el paso 1; Si se introduce “sortable = false” no se ordenará.
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Cada vez que se quiera dar la posibilidad de ordenar por una columna.

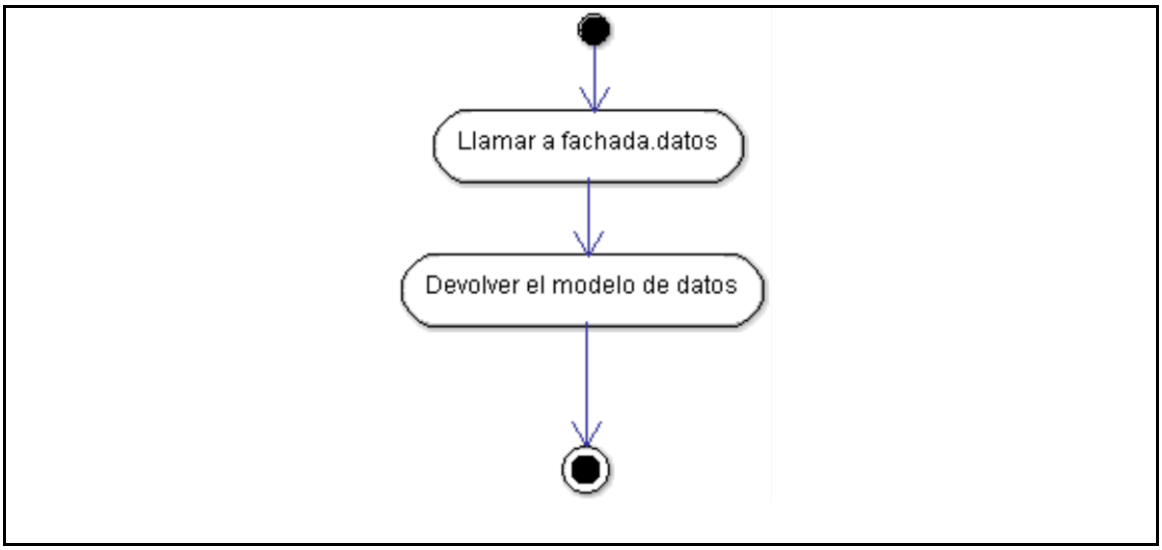
Nombre caso de uso:	**Control de paginado**
Descripción:	El usuario puede crear el control de paginado usando la etiqueta *CONTROLPAGINA.*
Lista de requisitos cubiertos:	• HTML extendido. • Control de paginado. • Contenido del control de páginas. • Página válida. • Extender contenido.

Precondiciones:	• Tener una etiqueta <angrid> como padre.
Postcondiciones:	• Se creará un control para cambiar de página.
Escenario principal:	1) Añadir etiqueta <controlpagina>. 2) **Añadir control de paginado al grid.**
Escenarios alternativos:	
Requisitos no funcionales:	• Usabilidad
Frecuencia:	Cada vez que se quiera añadir un control para las páginas.
	

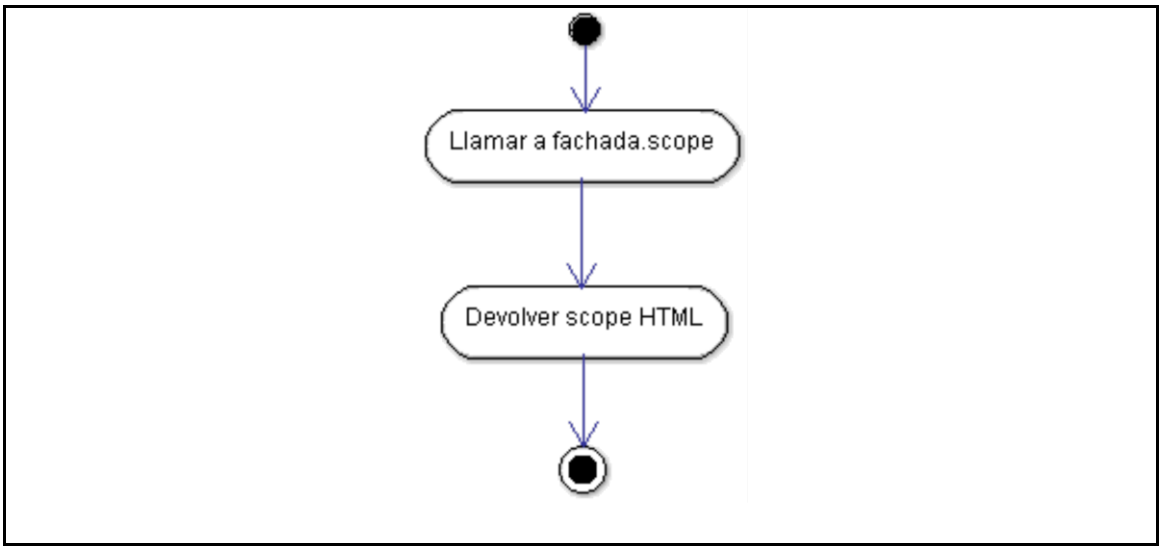
Nombre caso de uso:	**Control del grid a bajo nivel**
Descripción:	El usuario puede acceder a la clase del grid utilizando la fachada.
Lista de requisitos cubiertos:	• Acceso al grid.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Devolverá la clase grid.
Escenario principal:	1) Llamar a fachada.grid.. 2) **Devolver la clase grid.**
Escenarios alternativos:	
Requisitos no funcionales:	• Interoperabilidad

Frecuencia:	Cada vez que el desarrollador quiero manejar el grid a bajo nivel.
	

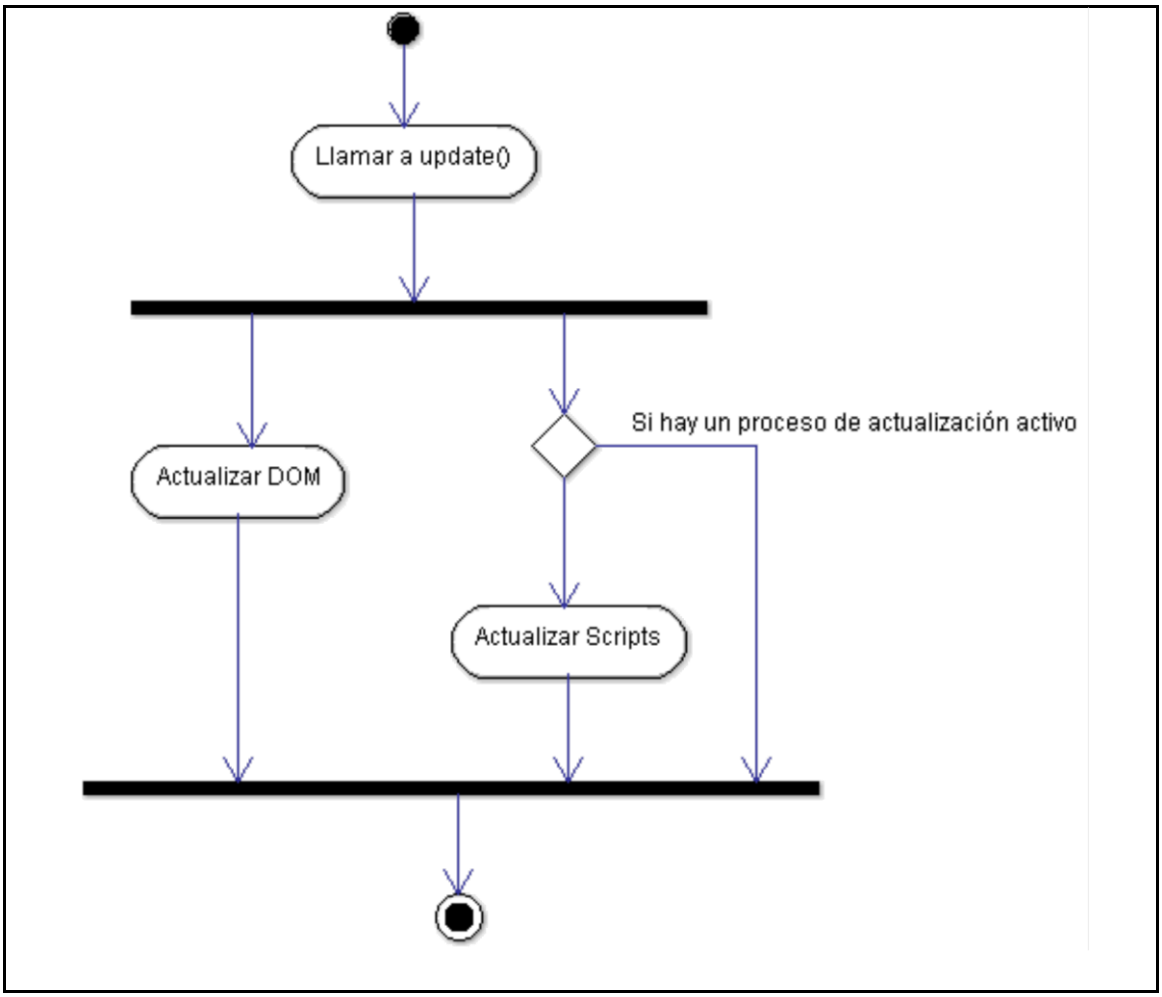
Nombre caso de uso:	**Acceso al modelo**
Descripción:	El usuario puede obtener el modelo de datos actualmente cargado en formato JSON a través de la fachada.
Lista de requisitos cubiertos:	• Acceso al modelo.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Devolverá el modelo de datos actual.
Escenario principal:	1) Llamar a fachada.datos. 2) **Devolver modelo de datos.**
Escenarios alternativos:	
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera acceder al modelo de datos cargado del grid.



Nombre caso de uso:	**Acceso al scope**
Descripción:	El usuario puede acceder al scope AngularJS del DOM del grid.
Lista de requisitos cubiertos:	• Acceso al scope.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Devolverá el scope HTML.
Escenario principal:	1) Llamar a fachada.scope. 2) **Devolver scope HTML.**
Escenarios alternativos:	
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera acceder al scope.

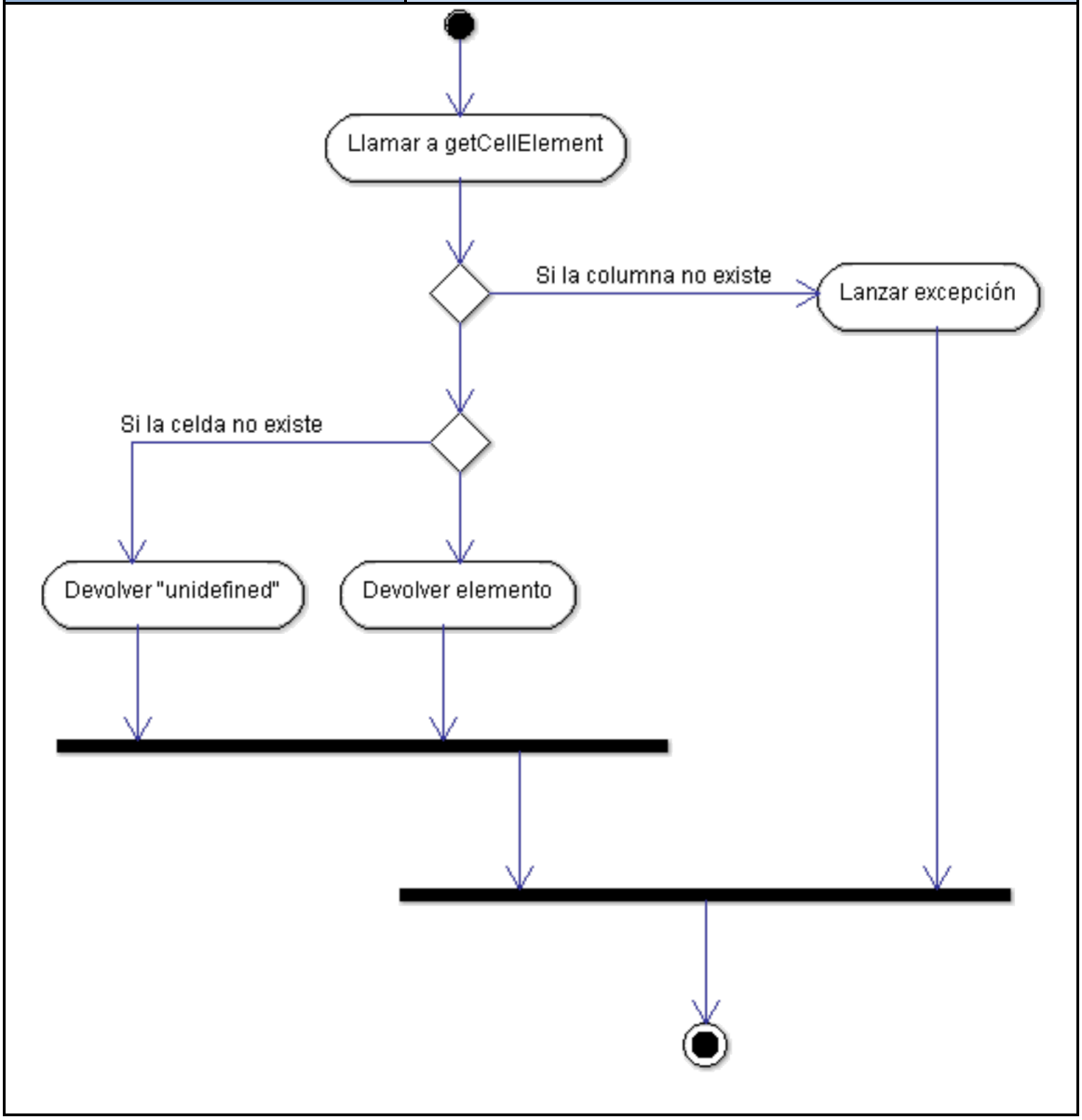


Nombre caso de uso:	**Actualizar**
Descripción:	El usuario puede actualizar la vista, el modelo o ambos.
Lista de requisitos cubiertos:	• Actualizar vista. • Actualizar DOM. • Actualizar.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Actualizará el grid.
Escenario principal:	1) Llamar a update(). 2) **Actualizar DOM.** 3) **Actualizar Scripts.**
Escenarios alternativos:	• En el paso 3; Si se estaba actualizando los scripts estos no se actualizarán.
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el usuario quiera actualizar el grid.



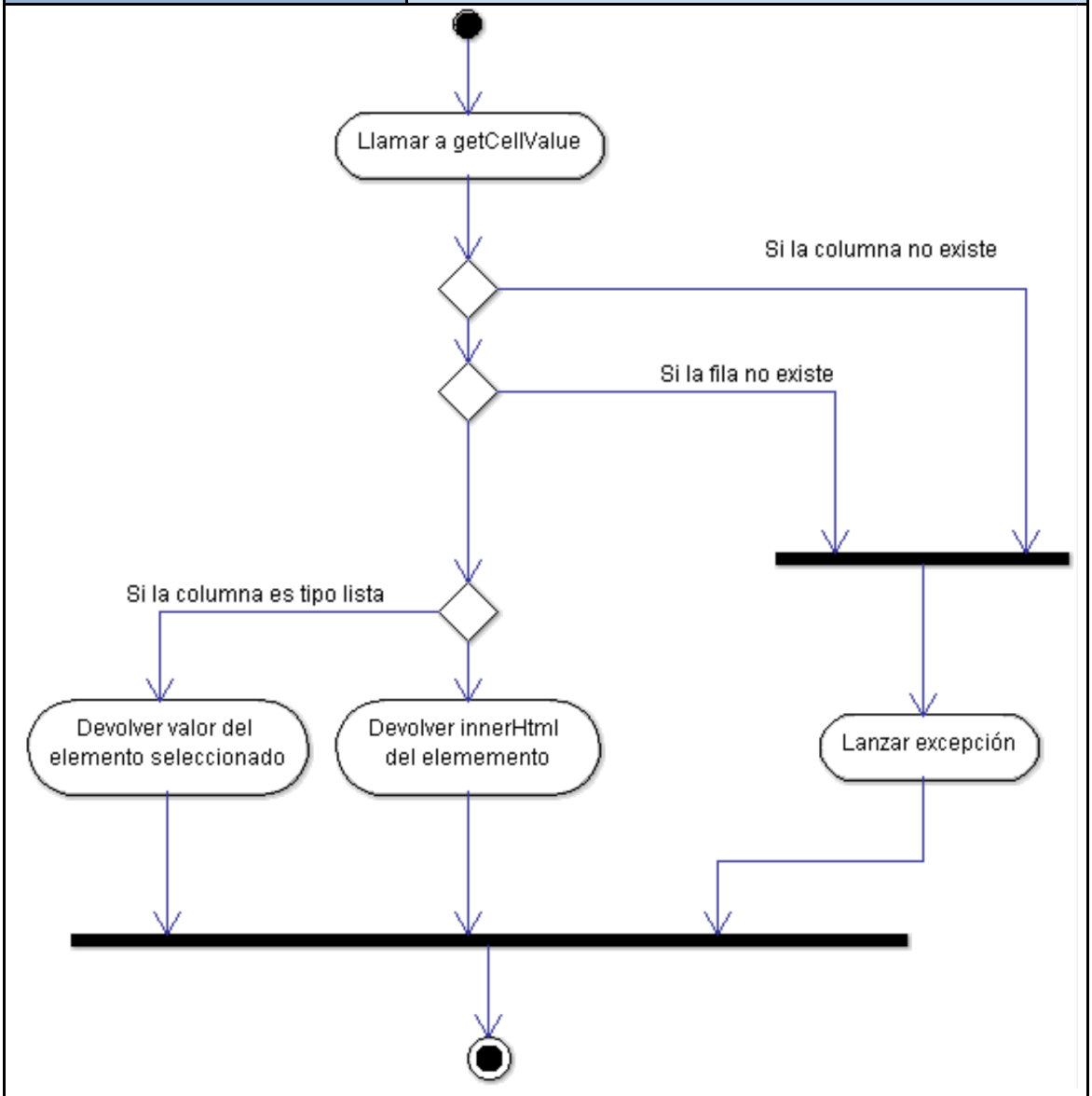
Nombre caso de uso:	**Obtener contenido de una celda del grid**
Descripción:	Permite obtener el objeto HTML de una celda.
Lista de requisitos cubiertos:	• Obtener una celda.
Precondiciones:	• Tener una instancia de la fachada. • Columna y fila deben ser números.
Postcondiciones:	• Devolverá el elemento HTML de la celda.
Escenario principal:	1) Llamar a getCellElement con el número de columna y de fila. 2) **Devolver el elemento HTML de esa celda.**
Escenarios alternativos:	• En el paso 1; Si la columna no existe se lanzará una excepción. • En el paso 1; Si la fila no existe se devolverá "undefined".

Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera obtener el contenido HTML de la celda.

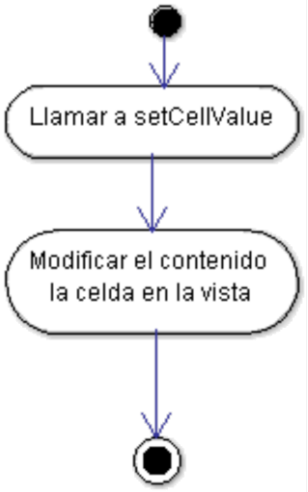


Nombre caso de uso:	**Obtener HTML de una celda del grid**
Descripción:	Permite obtener el valor en HTML de una celda de la tabla.
Lista de requisitos cubiertos:	• Obtener valor de la vista.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Devolverá el contenido HTML de la celda.

Escenario principal:	1) Llamar a getCellValue con la fila y la columna a obtener. 2) **Obtener elemento HTML de la celda.** 3) **Devolver el contenido HTML del elemento.**
Escenarios alternativos:	• En el paso 1; Si la columna no existe se lanzará una excepción. • En el paso 1; Si la fila no existe se producirá una excepción. • En el paso 3; Si la columna es una lista devolver el valor del elemento seleccionado.
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera obtener el contenido HTML de una celda.

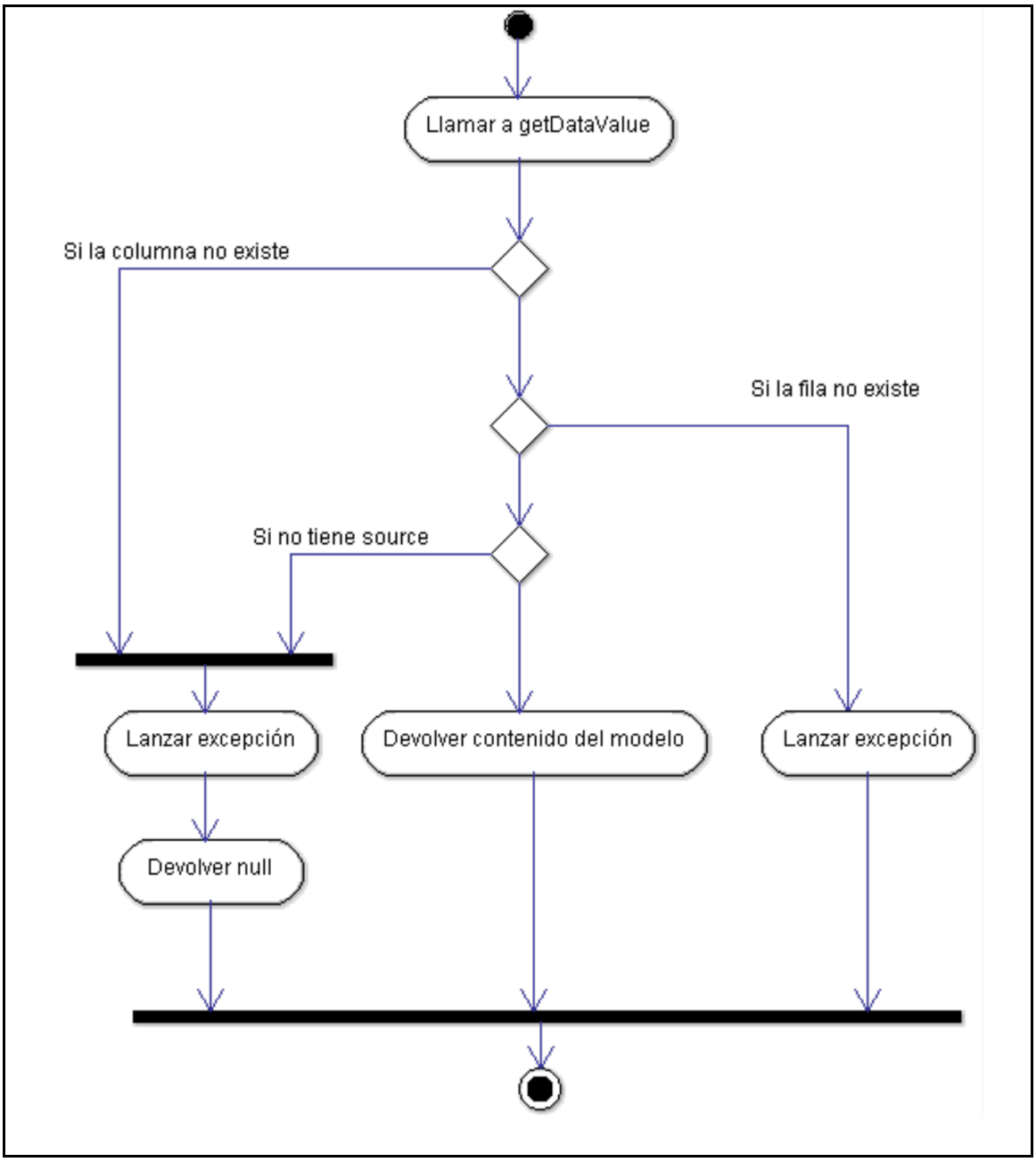


Nombre caso de uso:	**Establecer HTML de una celda del grid**
Descripción:	Permite establecer el valor HTML de una celda de la tabla.
Lista de requisitos cubiertos:	• Establecer valor de la vista.
Precondiciones:	• Tener una instancia de la fachada. • La columna y la fila deben ser valores numéricos.
Postcondiciones:	• Cambiará el contenido HTML de la celda.
Escenario principal:	1) Llamar a setCellValue con la fila, columna y nuevo contenido HTML. 2) **Modificar el contenido de la celda.**
Escenarios alternativos:	
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera cambiar el HTML de una celda.



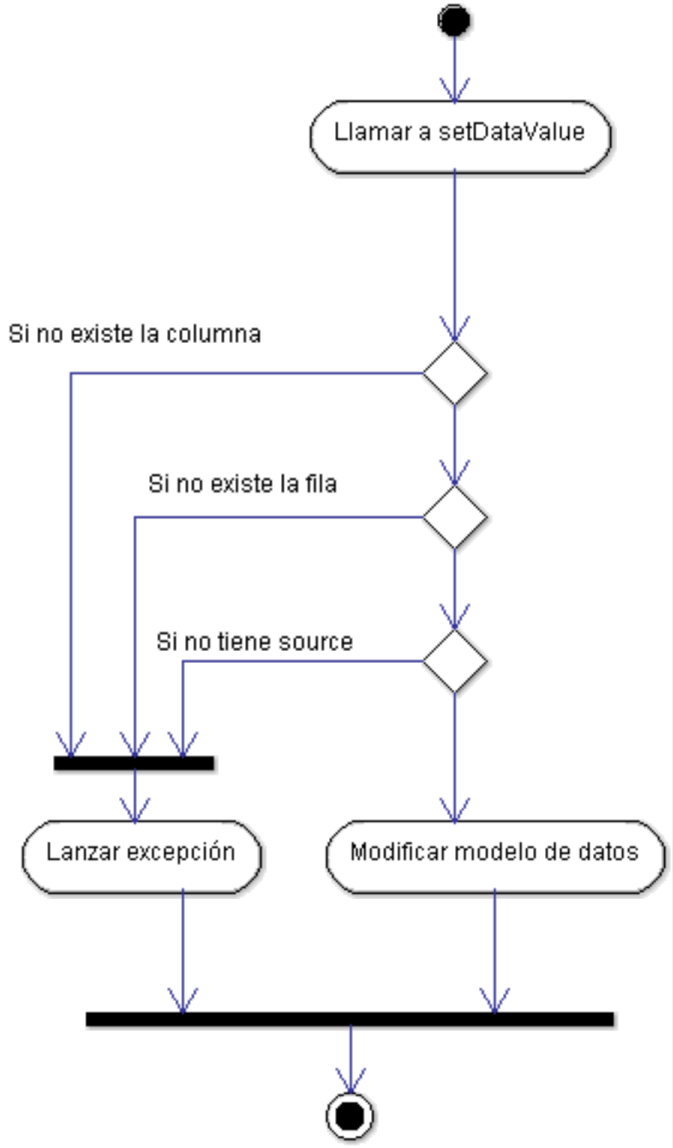
Nombre caso de uso:	**Obtener datos de una celda**
Descripción:	Permite obtener el valor de la fuente de datos relacionado a una celda.
Lista de requisitos cubiertos:	• Obtener valor del modelo.
Precondiciones:	• Tener una instancia de la fachada.

Postcondiciones:	• Obtiene el valor de una celda.
Escenario principal:	1) Llamar a getDataValue. 2) **Devolver valor del modelo.**
Escenarios alternativos:	• En el paso 1; Si no existe la columna se producirá una excepción y devolverá null. • En el paso 1; Si la columna no tiene atributo source se producirá una excepción y devolverá null. • En el paso 1; Si no existe la fila se producirá una excepción.
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera obtener el valor del modelo en una celda.

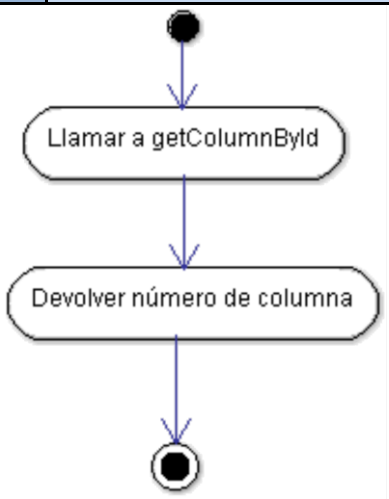


Nombre caso de uso:	**Establecer datos de una celda**
Descripción:	Permite establecer el valor de la fuente de datos relacionado a una celda.
Lista de requisitos cubiertos:	• Establecer valor del modelo.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Cambiará el valor de la celda en el modelo de datos.

Escenario principal:	1) Llamar a setDataValue con el número de fila y columna y el nuevo valor. 2) **Establecer el nuevo valor a la celda.**
Escenarios alternativos:	• En el paso 1; Si no existe la columna se producirá una excepción. • En el paso 1; Si la columna no tiene atributo source se producirá una excepción. • En el paso 1; Si no existe la fila se producirá una excepción.
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera modificar la fuente de datos de una celda.

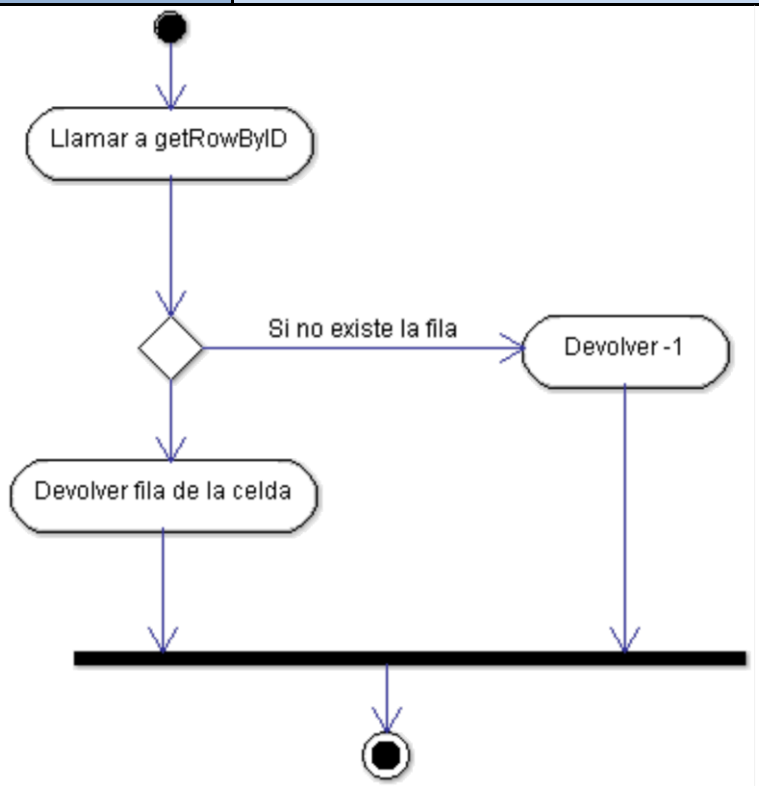


Nombre caso de uso:	**Buscar columna de una celda**
Descripción:	Permite obtener el número de columna en la que esta la celda.
Lista de requisitos cubiertos:	• Obtener columna por ID.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Devolverá el número de columna.
Escenario principal:	1) Llamar a getColumnById con el ID del elemento HTML de la celda. 2) **Devolverá el número de columna.**
Escenarios alternativos:	
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera obtener el número de columna de una celda.

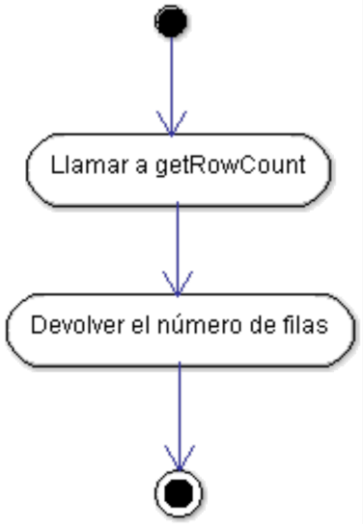


Nombre caso de uso:	**Buscar fila de una celda**
Descripción:	Permite obtener el número de fila en la que está una celda.
Lista de requisitos cubiertos:	• Obtener fila por ID.
Precondiciones:	• Tener una instancia de la fachada.

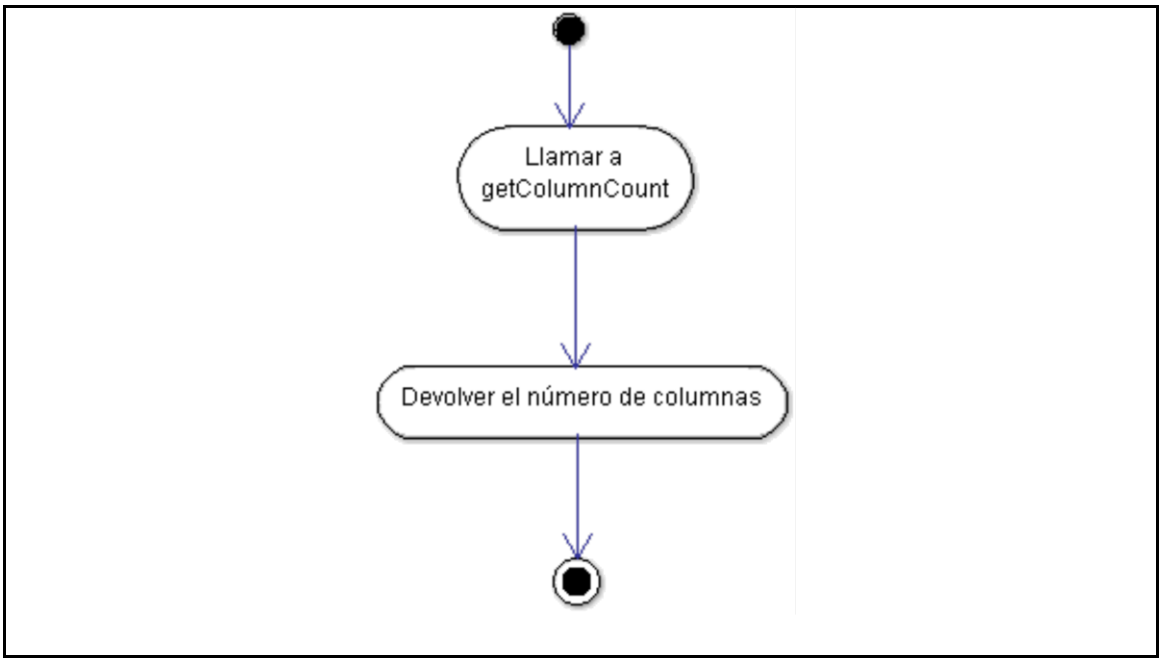
Postcondiciones:	• Devolverá el número de fila.
Escenario principal:	1) Llamar a getRowById con el ID del elemento HTML de la celda. 2) **Devolver el número de fila.**
Escenarios alternativos:	• En el paso 1; Si la fila no existe devolverá -1 .
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador necesite saber el número de fila en la que está una celda.



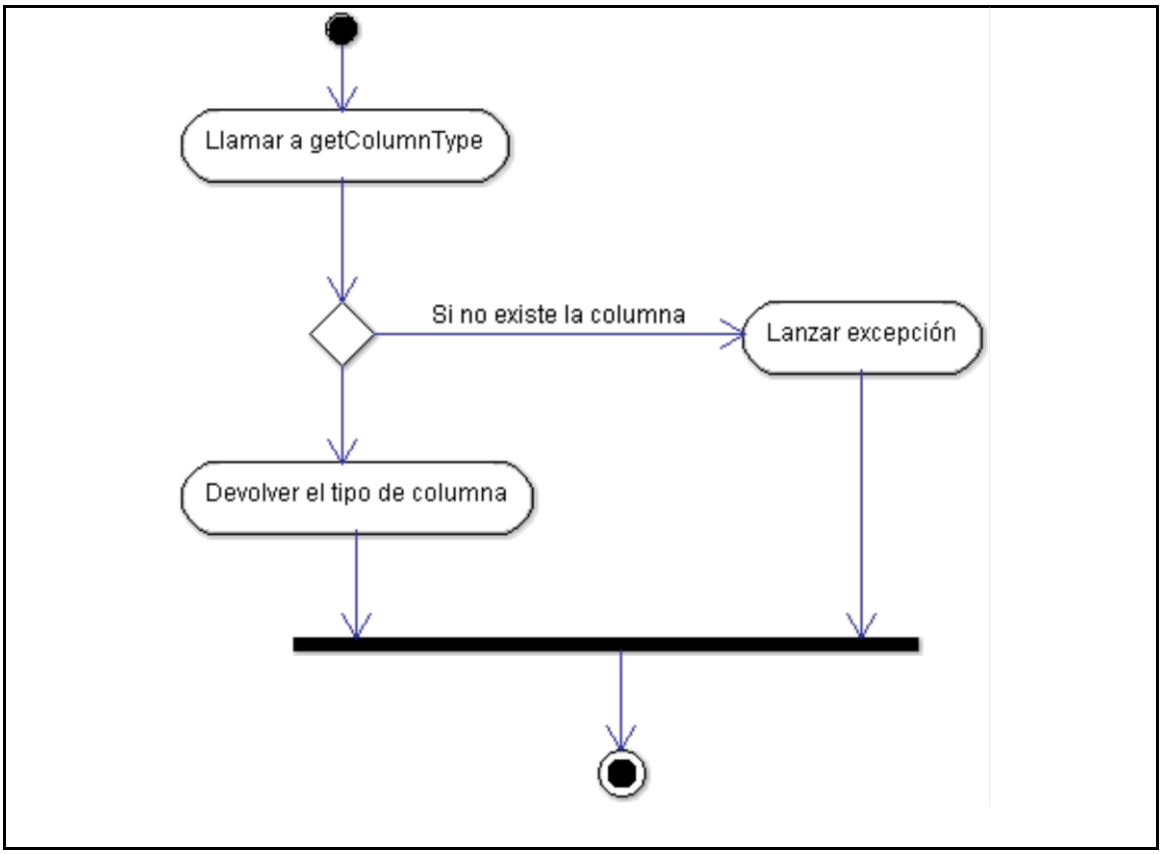
Nombre caso de uso:	**Obtener número de filas**
Descripción:	Obtiene la cantidad del filas existentes en el modelo.
Lista de requisitos cubiertos:	• Obtener número de filas.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Devuelve el número de filas.

Escenario principal:	1) Llamar a getRowCount. 2) **Devolver el número de filas.**
Escenarios alternativos:	
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador necesite saber el número total de filas.
	

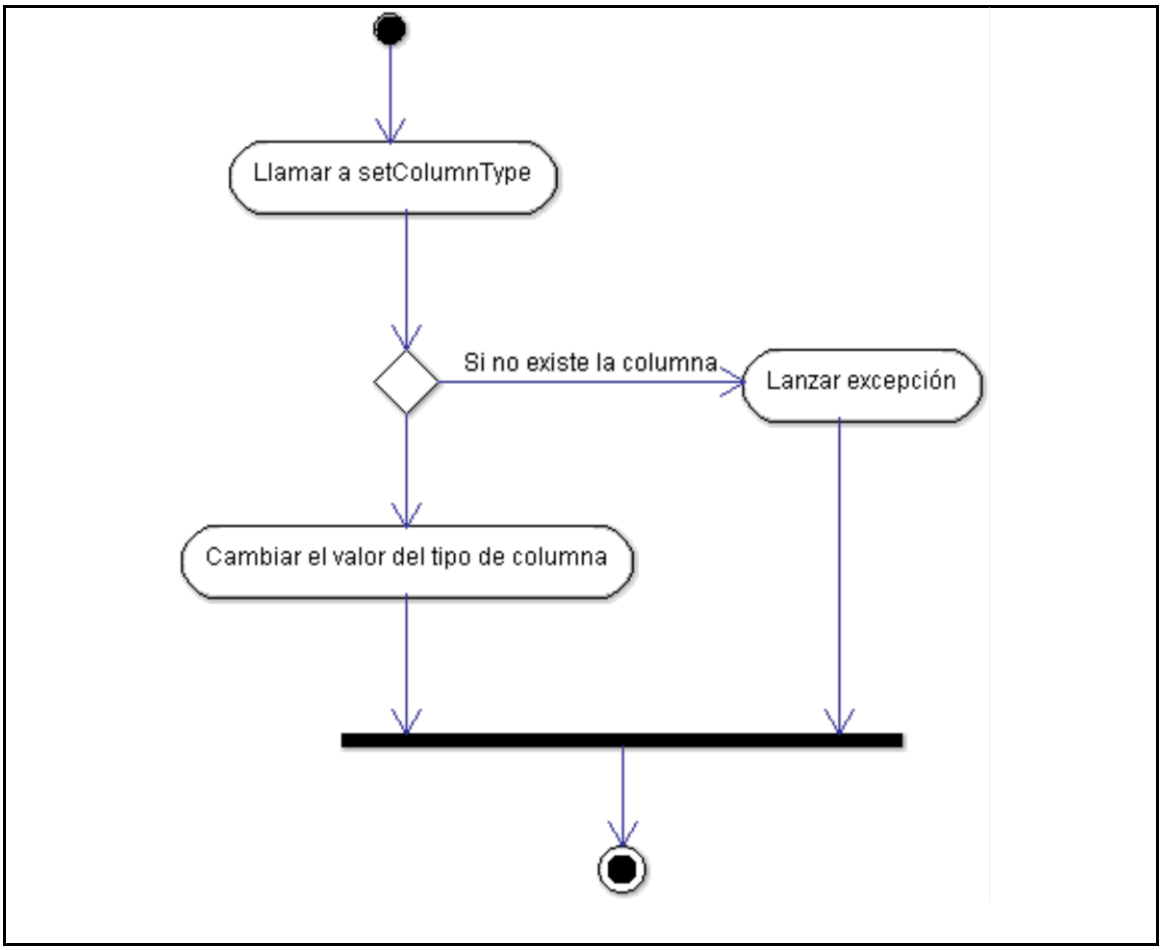
Nombre caso de uso:	**Obtener número de columnas**
Descripción:	Obtiene la cantidad del columnas existentes en el grid.
Lista de requisitos cubiertos:	• Obtener número de columnas.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Devuelve el número de columnas totales.
Escenario principal:	1) Llamar a getColumnCount. 2) **Devolver número de columnas.**
Escenarios alternativos:	
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador necesite saber el número de columnas configuradas en el grid.



Nombre caso de uso:	**Obtener tipo de columna**
Descripción:	Obtiene el texto identificativo del tipo de una columna.
Lista de requisitos cubiertos:	• Obtener el tipo de columna.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Devolverá el identificador de tipo de la columna.
Escenario principal:	1) Llamar a getColumnType con el número o nombre de columna. 2) **Devolver el tipo de columna.**
Escenarios alternativos:	• En el paso 1; Si no existe la columna introducida se producirá una excepción.
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera obtener el tipo de una columna.

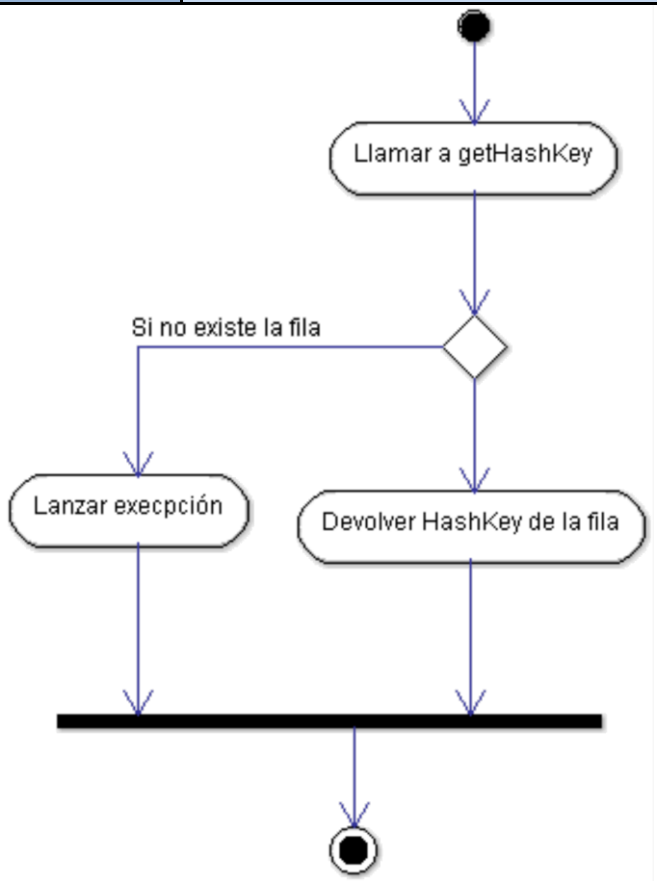


Nombre caso de uso:	**Establecer tipo de columna**
Descripción:	Establece el tipo de la columna seleccionada.
Lista de requisitos cubiertos:	• Establecer el tipo de columna.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Cambiará el tipo de columna. • Se debe actualizar el grid con update().
Escenario principal:	3) Llamar a setColumnType con el número o nombre de columna y el identificador de tipo. 4) **Cambiar el tipo de columna.**
Escenarios alternativos:	• En el paso 1; Si no existe la columna introducida se producirá una excepción.
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que se quiera cambiar la representación de los valores de las columnas.

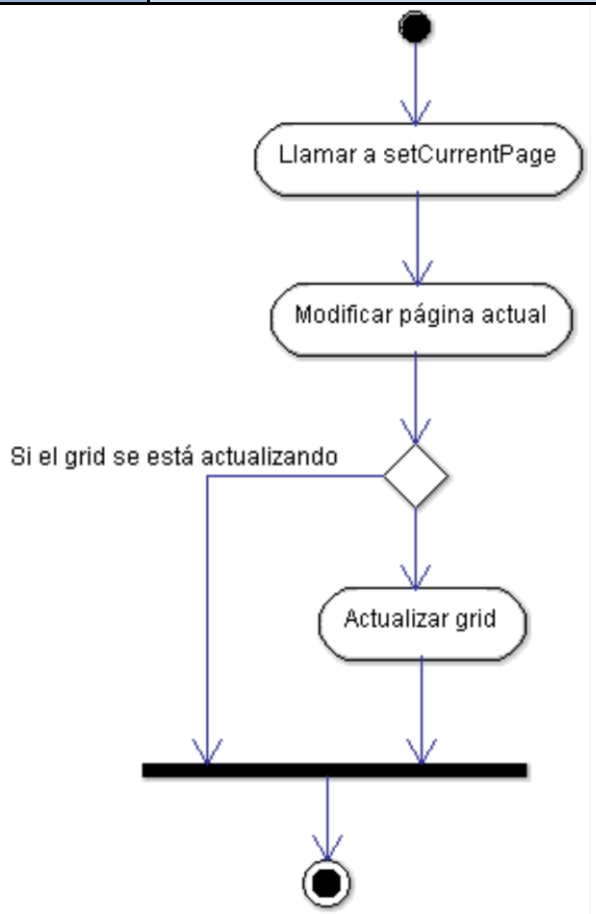


Nombre caso de uso:	**Obtener el ID único de fila**
Descripción:	Obtiene el HashID único asignado por AngularJS.
Lista de requisitos cubiertos:	• Obtener el ID único de una fila.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Devolverá el ID asignado por AngularJS.
Escenario principal:	1) Llamar a getHashKey con el número de fila. 2) **Devolver el HashKey de la fila.**
Escenarios alternativos:	• En el paso 1; Si se introduce un número de fila que no existe se producirá ArrayOutOfBoundException.
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera obtener el ID único

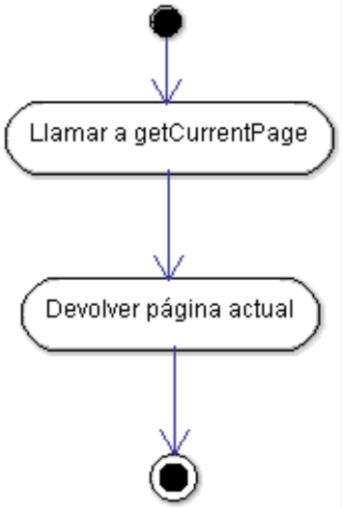
	de cada fila.



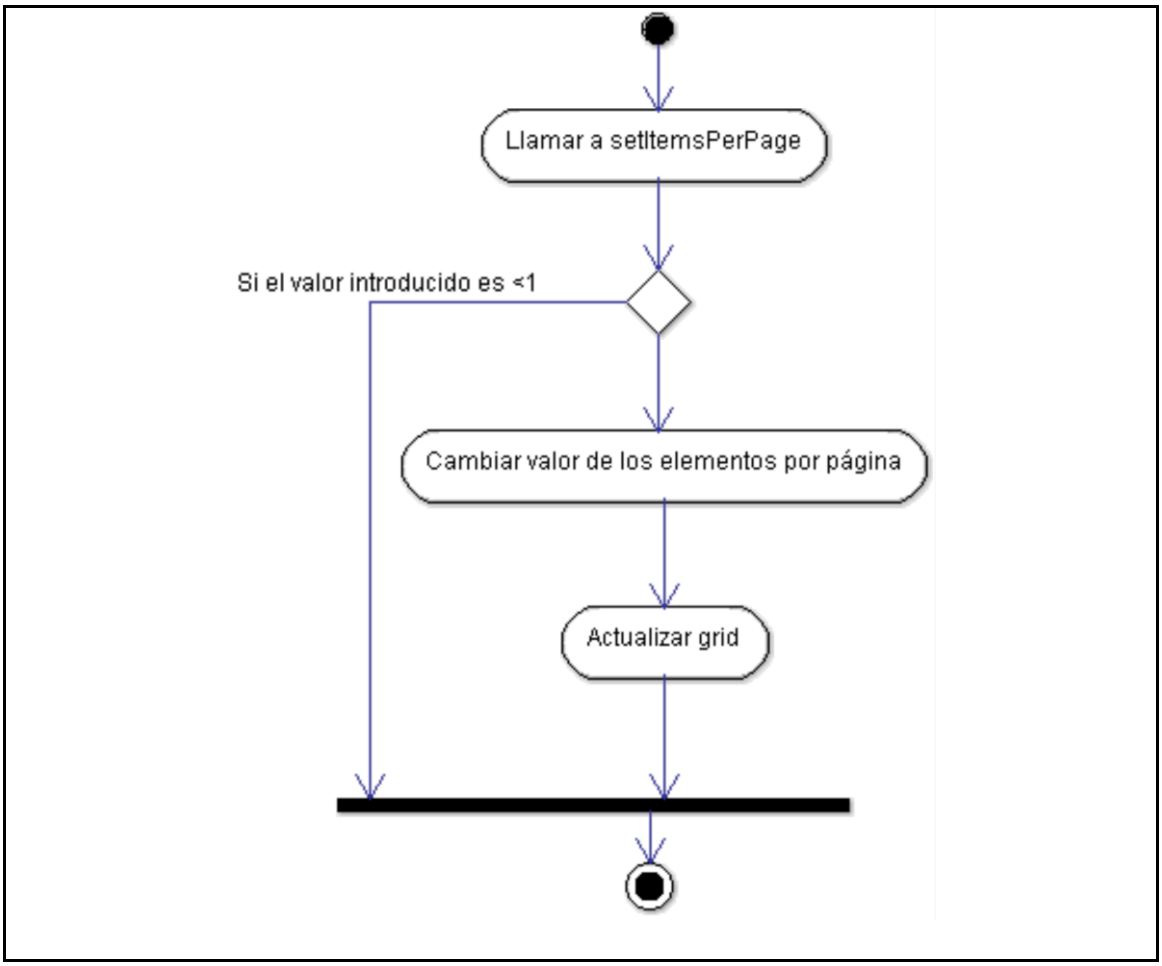
Nombre caso de uso:	**Establecer la página actual**
Descripción:	Establece el número de página que se mostrará actualmente.
Lista de requisitos cubiertos:	• Establecer la página actual.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Se visualizará la página introducida.
Escenario principal:	1) Llamar a setCurrentPage con la página a la que moverse. 2) **Establecer la página actual en el grid.** 3) **Actualizar grid.**
Escenarios alternativos:	• En el paso 3; Si se estaba actualizando el grid en ese momento no se actualizará el grid otra vez.

Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera cambiar de página manualmente.
	

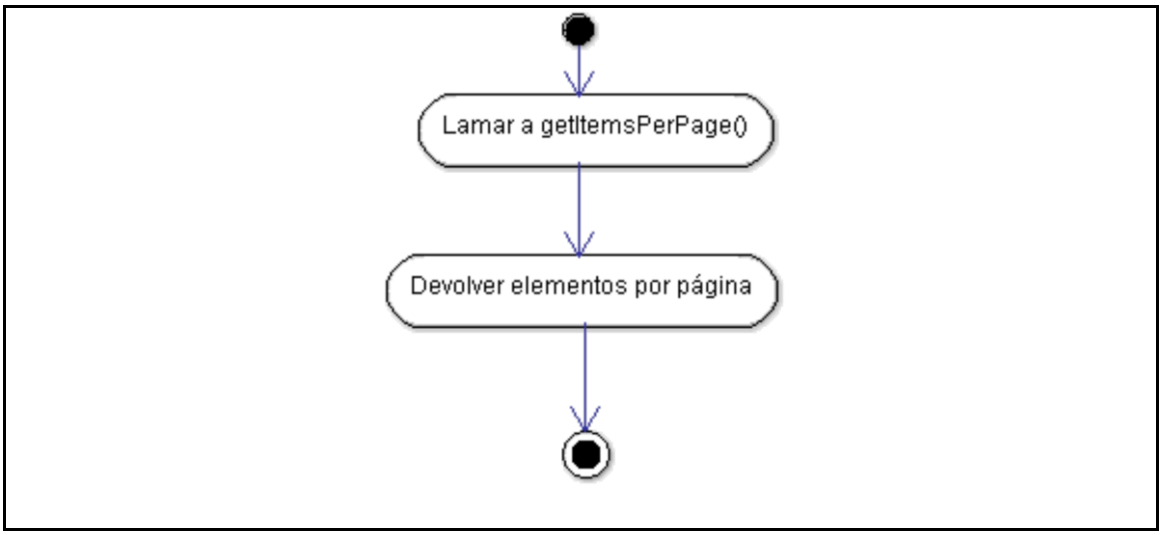
Nombre caso de uso:	**Obtener la página actual**
Descripción:	Obtiene el número de página que se mostrará actualmente.
Lista de requisitos cubiertos:	• Obtener la página actual.
Precondiciones:	• Tener una instancia del grid.
Postcondiciones:	• Obtiene la página que se muestra actualmente.
Escenario principal:	1) Llamar a getCurrentPage. 2) **Devolver la página actual.**
Escenarios alternativos:	

Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador necesite saber la página que se está visualizando actualmente.
	

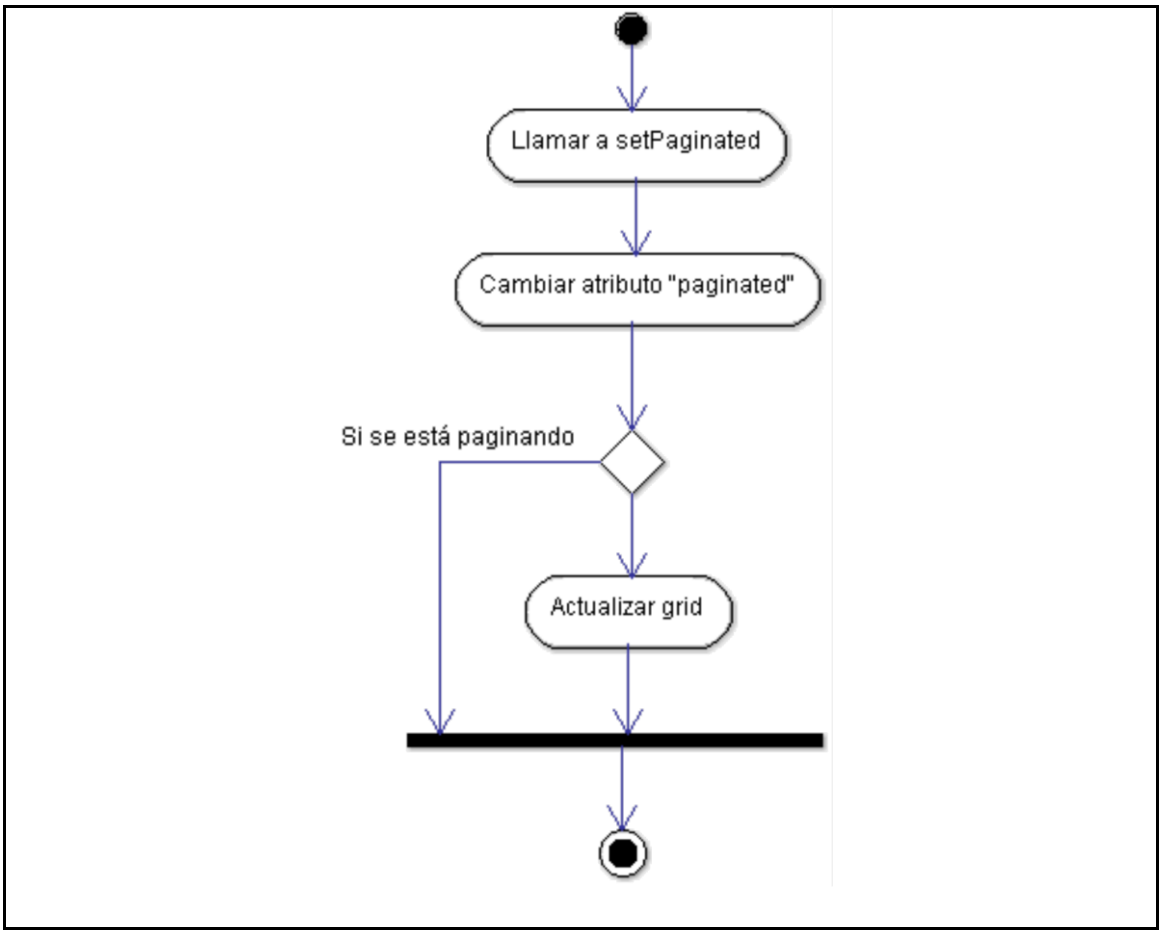
Nombre caso de uso:	**Establecer elementos por página**
Descripción:	Establece el número de elementos que se mostrarán en cada página del grid paginado.
Lista de requisitos cubiertos:	• Establecer elementos por página.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Se mostrarán los elementos por página determinados en el grid.
Escenario principal:	1) Llamar a setItemsPerPage. 2) **Cambiar valor del los elementos por página.** 3) **Actualizar grid.**
Escenarios alternativos:	• En el paso 1; Si se introduce un número menor que 1 se producirá una excepción.
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera cambiar el número de elementos que se muestran en cada página.



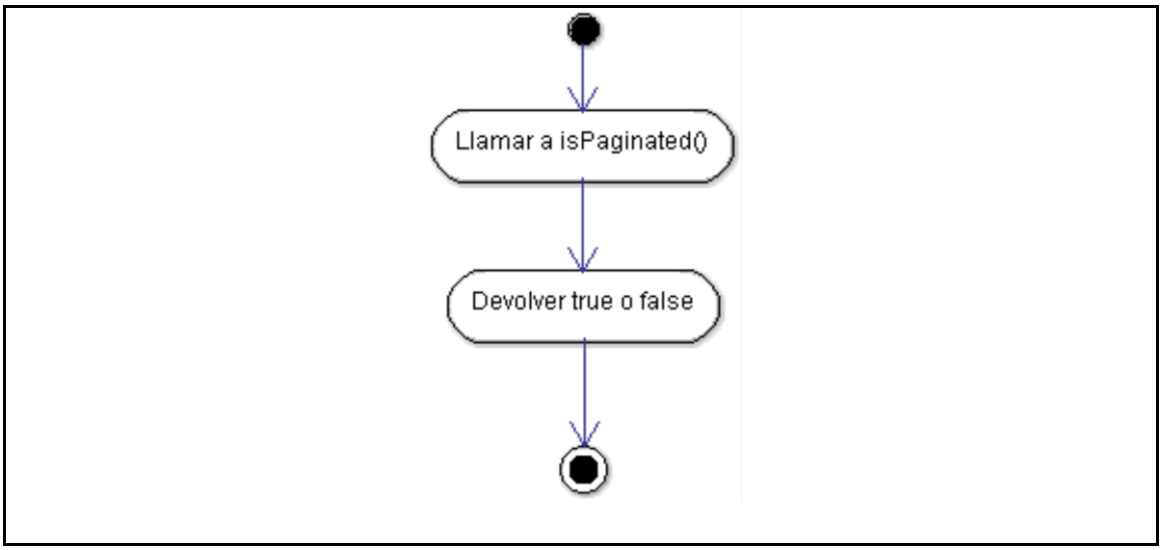
Nombre caso de uso:	**Obtener elementos por página**
Descripción:	Obtiene el número de elementos que se muestran en cada página del grid paginado.
Lista de requisitos cubiertos:	• Obtener elementos por página.
Precondiciones:	• Tener una instancia del grid.
Postcondiciones:	• Devolverá el número de elementos por página.
Escenario principal:	1) Llamar a getItemsPerPage(). 2) **Devolver los elementos por página.**
Escenarios alternativos:	
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera saber la cantidad de elementos por página que se están mostrando.



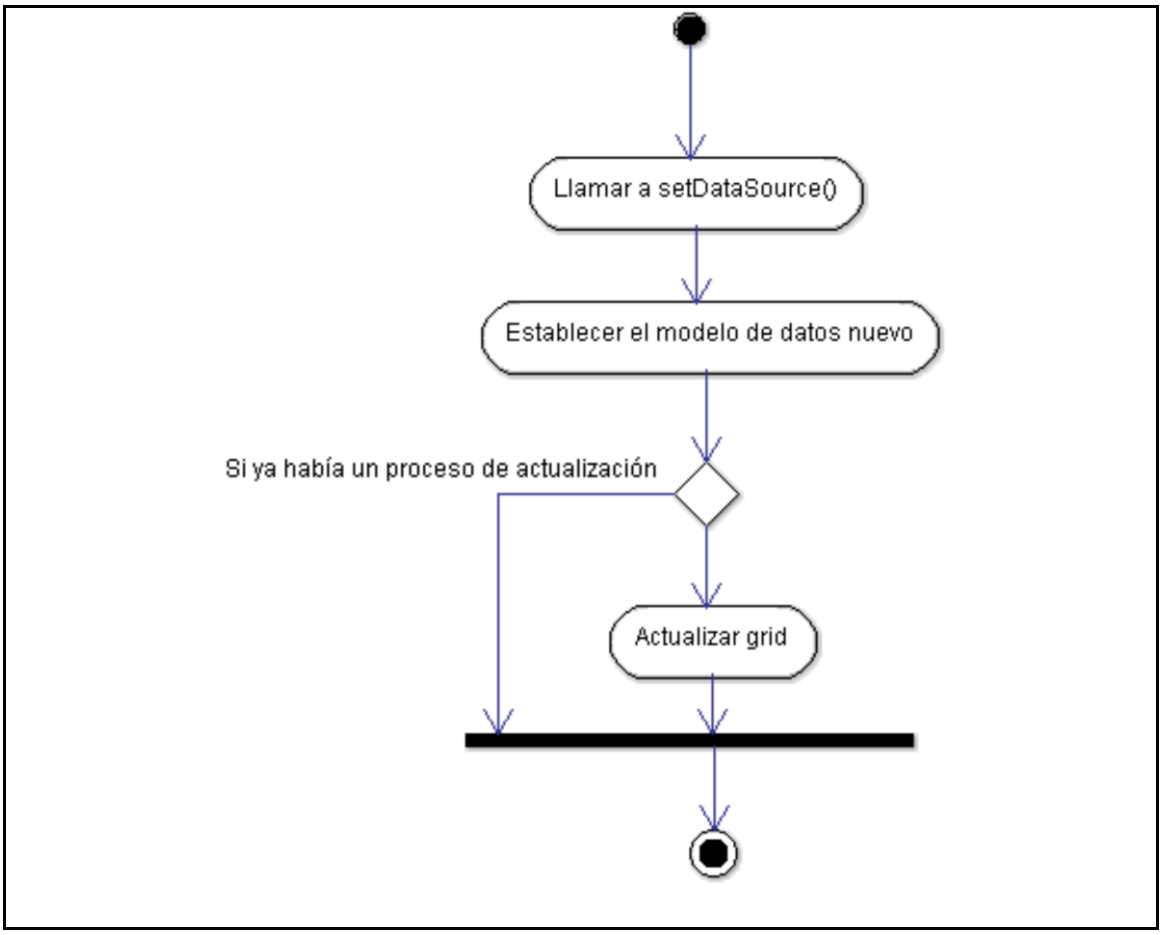
Nombre caso de uso:	**Establecer paginación**
Descripción:	Establece si se debe paginar el grid.
Lista de requisitos cubiertos:	• Establecer paginación.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• El grid será paginado ahora.
Escenario principal:	1) Llamar a setPaginated indicando si se quiere paginar. 2) **Cambiar atributo "paginated".** 3) **Actualizar grid.**
Escenarios alternativos:	• En el paso 3; Si ya se estaba actualizando la vista no se volverá a actualizar.
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera paginar o dejar de paginar el grid.



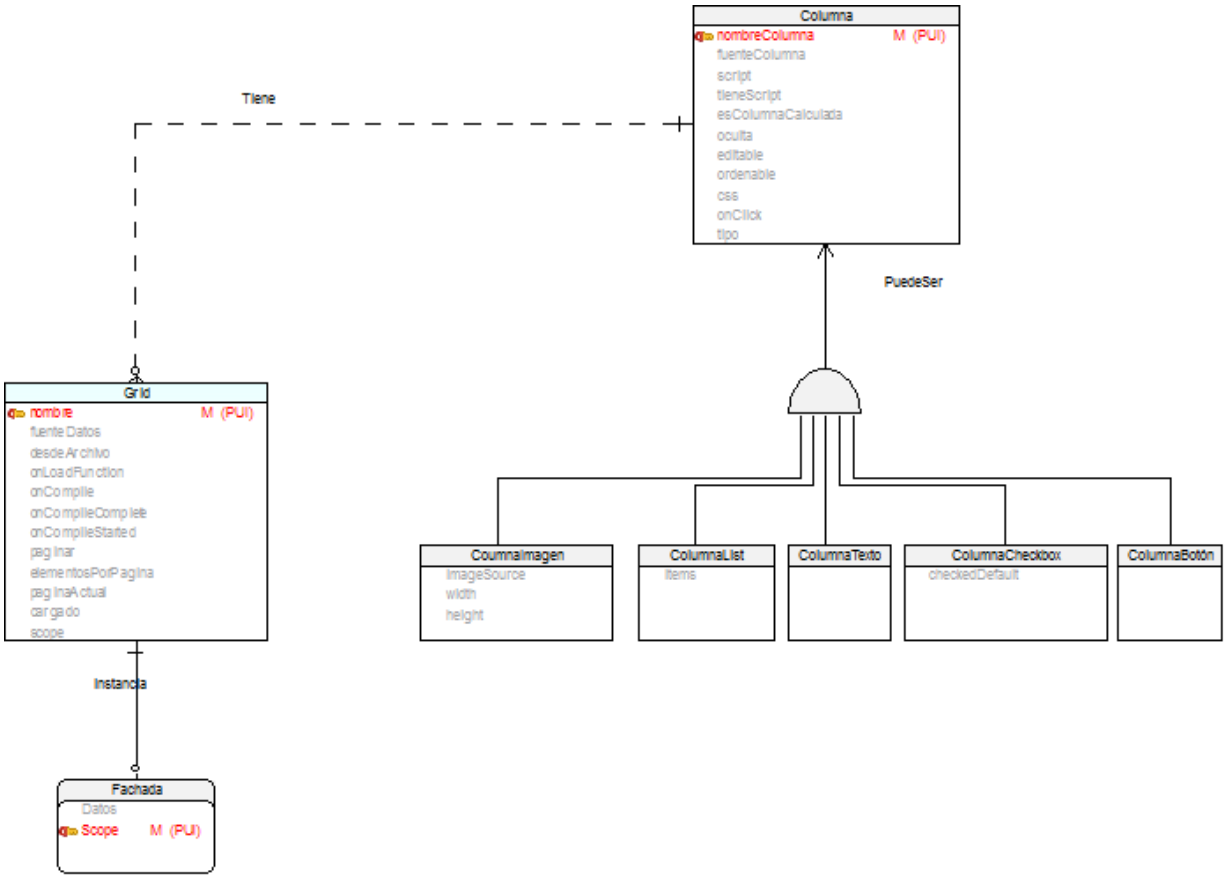
Nombre caso de uso:	**Obtener paginación**
Descripción:	Obtiene si se está paginando el grid.
Lista de requisitos cubiertos:	• Obtener paginación.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Devolverá si se está paginando el grid.
Escenario principal:	1) Llamar a isPaginated(). 2) **Devolver true o false si se está paginando.**
Escenarios alternativos:	
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el desarrollador quiera saber si se está paginando el grid.



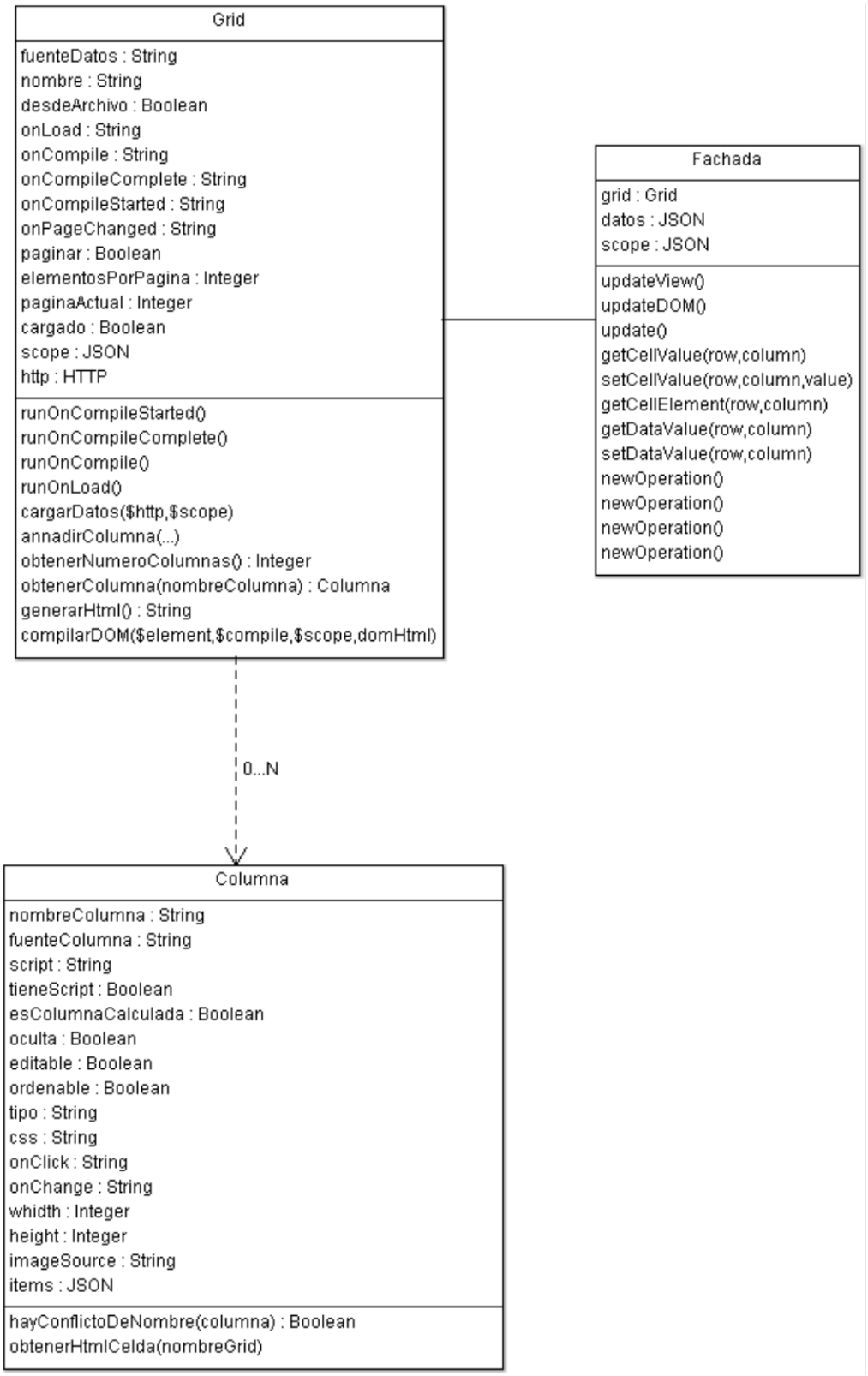
Nombre caso de uso:	**Cambiar la fuente de datos**
Descripción:	Cambia el modelo de datos actual del grid por otro.
Lista de requisitos cubiertos:	• Cambiar la fuente de datos.
Precondiciones:	• Tener una instancia de la fachada.
Postcondiciones:	• Se habrá cambiado la fuente de datos.
Escenario principal:	1) Llamar a changeDataSource () de la fachada. 2) **Cambiar el modelo de datos local.** 3) **Actualizar grid.**
Escenarios alternativos:	• En el paso 3; Si ya se estaba actualizando la vista no se volverá a actualizar.
Requisitos no funcionales:	• Interoperabilidad
Frecuencia:	Cada vez que el usuario quiera cambiar de fuente de datos.


Llamar a setDataSource()
Establecer el modelo de datos nuevo
Si ya había un proceso de actualización
Actualizar grid

4.7. Modelo Entidad-Relación



4.8. Diagrama de clases



5. Conclusiones

La aplicación desarrollada en este TFG está orientada al desarrollo de aplicaciones más complejas. Forma un control unitario, una herramienta para desarrolladores de aplicaciones web para mostrar y manejar datos desde JSON.

El mayor inconveniente existente con este proyecto es el uso de la librería AngularJS y sus aplicaciones, así como la imposibilidad de poner funciones JavaScript directamente en los atributos de eventos y scripts.

La parte más compleja de este proyecto ha sido el proceso de aprendizaje de AngularJS; complejo, abrupto y en ocasiones frustrante. En concreto, la compilación del código HTML y ejecución de los scripts de usuarios tras compilar y actualizar el DOM han sido los apartados más difíciles del proyecto.

Los mayores problemas que se han encontrado ha sido la compilación asíncrona del código lo que producía un solapamiento en la ejecución de los scripts y ejecución de los scripts. Se solventó utilizando un temporizador y la recomendación de insertar las funciones JavaScript antes del grid.

Otro de las barreras con las que se ha encontrado en el desarrollo de este proyecto ha sido la carga desde un fichero del JSON. Tras muchas pruebas se concluyó que, por motivos de seguridad, no se podrían cargar los ficheros JSON en un entorno de ejecución local (ejecutando la página web desde local).

Finalmente, al término del desarrollo se encontró un defecto difícil de solucionar. AngularJS asignaba un ID no secuencial a cada fila. Se ha tenido que proporcionar una interfaz para calcular el número de fila con respecto al ID que proporciona AngularJS.

En cuanto a documentación, el mayor problema ha surgido en los artefactos relacionados con Ingeniería del Software. No se tenía experiencia en hacer Ingeniería del Software para aplicaciones no destinadas al usuario final. Ha sido complicado reconocer los actores que manejarán el producto final y como interactuarían con el software.

Aunque, en conclusión, se han adquirido muchos conocimientos nuevos sobre el MVC, JavaScript, jQuery, HTML y CSS, del DOM HTML y, por supuesto, AngularJS.

6. Trabajo futuro

Aunque Angrid sea completamente funcional sí que, en un futuro, se le puede añadir funcionalidades que mejorarían aún más su usabilidad.

Carga desde un servidor o servicio

El objetivo de Angrid es mostrar datos cargados desde un JSON que se encuentre en un archivo o en una variable JavaScript. Por lo generar estos datos serán obtenidos desde una base de datos o un servicio alojado en una web o servidor.

Por ello, la carga de un JSON desde un archivo o una variable limita al programador y le fuerza a obtener estos datos de antemano por otros medios. En este punto, el programador se vería obligado a utilizar, por ejemplo, AJAX para obtener este fichero JSON desde un dominio externo.

Esta situación, que es bastante común en una aplicación web, se podría complicar debido a este contratiempo.

Por ello, uno de los trabajos futuros que se podría implementar sería el dar soporte a Angrid para cargar la fuente de datos desde un servidor externo o servicio REST. Esto le permitiría obtener la información que se mostrará en el grid desde una base de datos orientado a ficheros como MongoDB o CouchDB.

Respecto a la conexión a bases de datos, también se podría extender a otros tipos de bases de datos relacionales como MySQL o PosgreSQL permitiendo consultas SQL predefinidas.

Estas consultas SQL pueden requerir lenguajes de servidor como Java o PHP, llamados mediante JavaScript, o bien, se puede realizar la conexión mediante NodeJS. Esto nos permitiría realizar filtros avanzados desde el origen de datos.

Paginación desde una fuente externa

Otro punto mejorable del proyecto sería la paginación. La paginación en Angrid es automática respecto a los datos que están actualmente cargados, es decir, las N tuplas de datos cargadas actualmente en el grid serán divididas en P páginas.

El problema de que se paginen los datos de esta manera es que, si los datos se cargan desde un servidor o un servicio web, normalmente los datos no se cargan en su totalidad. Esto quiere decir que los datos vendrán paginados ya desde el origen.

Esta complicación en la paginación se ha solucionado actualmente permitiendo asignar un número de páginas manualmente al control de paginación y, al cambiar de página, se notifica mediante un evento del grid la página a la que se ha cambiado.

Cuando el grid notifique un cambio de página el programador deberá cargar el nuevo modelo de datos desde el servidor y actualizar la vista de datos del grid.

Por ello, otra propuesta de mejora es resolver este problema de una forma más sencilla y eficaz para el programador.

Algunas ideas para resolver este problema son la posibilidad de determinar el número total de registros o de crear un modelo de datos JSON, cuyo contenido sea igual al número de registros de la fuente y se vaya rellenando este modelo con los datos cargados del origen cuando sea necesario.

Operaciones comunes con los registros

En cuanto a visualización de datos Angrid funciona muy bien, sin embargo su manejo puede ser complicado. Una operación tan simple como eliminar un registro conllevaría al programador a crear una columna con un control que al pulsarlo eliminase o sobrescribiese el modelo de datos actual.

Otra mejora muy importante sería precisamente la que resolvería estos problemas del manejo de los datos desde el grid.

Operaciones como la eliminación de registros o creación de nuevos son fácilmente implementables.

Filtros y validación de cambios

Una operación que si está implementada es la de edición de registros mediante la propiedad “editable”. Esta propiedad permite modificar en la mayoría de los casos los datos de las columnas de cada registro.

Sin embargo, tras finalizar la edición del dato el cambio se guarda en el modelo automáticamente. Esto está bien para la mayoría de tipos de datos, pero para algunos, como fechas, emails, o un DNI, un cambio aleatorio o incontrolado de su contenido no es aceptable.

Se le debe permitir al programador añadir un filtro para un campo y/o permitir validar los cambios después de que el usuario final los edite y revertirlos si es necesario. Quizás un evento de modificación que permitiese abortar los cambios sería una solución.

Nuevos tipos de columnas

Actualmente, Angrid permite los tipos básicos de columna para el funcionamiento del grid en general, pero tipos de columnas nuevos que implementasen campos de tipo email, fecha, o color serían muy útiles.

La implementación de estas columnas nuevas podría corresponder con los tipos de campos de entrada nuevos en HTML5, por lo que la coincidencia llevaría a la aplicación más cerca si cabe de HTML5.

7. Presupuesto

En este apartado se estima el coste de este proyecto. Para ello nos hemos valido de técnicas de Gestión de Proyectos como son IFPUG y COCOMO.

Mediante IFPUG calcularemos los puntos de función que más adelante utilizaremos en COCOMO para el cálculo del esfuerzo y costes. (10)

Lo primero de todo será obtener la complejidad de los ficheros de la aplicación:

Fichero	DET	RET	Complejidad	PFs
Grid	15	2	Baja	7
Fachada	2	1	Baja	7

En cuanto al fichero Grid vemos que está compuesto por 15 DET que corresponderían con los valores que debemos mantener en el fichero (fuente, nombre, hojas de estilo, etc.). En cuanto a los RET sabemos que el grid está formado por columnas por ello se debe contar como RET a Grid y a Columna.

Ahora que tenemos los puntos de función que forman los datos de la aplicación, deberemos calcular los puntos de función de las procedimientos que mantienen o consumen estos datos. Se deben tener en cuenta todas las funciones que traspasen la frontera de la aplicación.

Función	Tipo	DET	FTR	Complejidad	PFs
Cargar el modelo	INPUT	3	1	Baja	3
Compilar vista	OUTPUT	4	2	Baja	4
Añadir columna	INPUT	17	1	Media	4
Modificar modelo	INPUT	3	1	Baja	3

Modificar columna	INPUT	2	1	Baja	3
Obtener columnas	QUERY	0	1	Baja	3
Obtener filas	QUERY	0	1	Baja	3
Obtener modelo	QUERY	0	1	Baja	3

Ahora ya tenemos los puntos de función no ajustados de los ficheros y de las funciones que traspasan la frontera de la aplicación.

En total tenemos: $PFNA = (7 + 7) + (3 + 4 + 4 + 3 + 3 + 3 + 3 + 3) = 40$

Para ajustar estos puntos de función deberemos obtener el Factor de Complejidad Técnica que se calcular teniendo en cuenta unos parámetros cuyo valor irá de 0 a 5.

Parámetro	**Valor (0 – 5)**
Comunicación de datos	2
Actualización en línea	4
Servicios distribuidos	4
Procesamiento complejo	1
Desempeño	2
Reusabilidad	4
Uso sobrecargado	0
Facilidad de instalación	5
Rata de transacciones	0

Facilidad de operación	5
Entrada de datos en línea	1
Múltiples lugares de operación	3
Eficiencia del usuario final	5
Facilidad de modificación	4
TOTAL	**38**

En total vemos que $\rho = \sum parametros = 38$

Ahora podemos calcular el Factor de Complejidad Técnica:

$$FCT = 0.65 + (0.01 * \rho) = 0.65 + (0.01 * 38) = 1.03$$

Finalmente calculamos los puntos de función ajustados:

$$PFA = PFNA * FCT = 40 * 1.03 = 41.2$$

Ahora que tenemos los puntos de función podremos utilizar COCOMO, sin embargo, antes deberemos estimar las líneas de código finales que tendrá el proyecto.

Como no existe ningún baremo de líneas de código por punto de función para AngularJS se ha decidido utilizar su lenguaje más próximo, que en este caso es JavaScript.

Las líneas de código de JavaScript, según *QSM Function Points Languages Table*, de media, son 42 líneas de código por punto de función. (11) (12)

Por lo tanto: $LOC = 41.2 * 42 = 1730.4$

Es importante tener en cuenta que el baremo utilizado es de JavaScript puro, por lo que las líneas de código para AngularJS reales serían menores que las calculadas aquí, ya que AngularJS permite efectivamente esto, reducir líneas de código.

Por ello redondearemos las líneas de código hacia abajo: $MLOC = 1$

Ahora tenemos todo lo necesario para calcular el esfuerzo del proyecto mediante COCOMO. Utilizaremos COCOMO con el modelo intermedio. Este modelo nos exige que calculemos unos valores de ajuste para el cálculo del esfuerzo.

Factor de ajuste	Valor
RELY	0.75
DATA	0.94
CPLX	1
TIME	1
STOR	1
VIRT	1.30
TURN	1
ACAP	0.86
AEXP	1.29
PCAP	1
VEXP	1.21
LEXP	1.14
MODP	0.91
TOOL	1.10
SCED	1

Estos valores multiplicados nos dan el factor de ajuste: $EAF = 1.403$

Ahora estamos listos para calcular el esfuerzo:

$$E = \alpha * MLOC^{\beta} * EAF$$

Los parámetros alfa y beta vienen determinados por el tipo de sistema; este puede ser orgánico, semilibre, rígido. En este caso, se ha decidido definir este sistema como rígido, ya que se trata de un proyecto con restricciones marcadas por el ámbito del proyecto y existen problemas técnicos en el mismo, que no pueden resolverse con la experiencia del desarrollador ya que no la tiene con esta tecnología.

Con este sistema vemos que Alfa = 2.6, Beta = 1.20, Gamma = 2.50 y Delta = 0.32

$$E = 2.6 * 1^{1.2} * 1.403 = \mathbf{3.64\ persona - mes}$$

El tiempo de desarrollo y las personas involucradas sería:

$$TDev = \gamma * E^{\delta} = 2.50 * 3.64^{0.32} = 3.78$$

$$P = \left\lceil \frac{E}{TDev} \right\rceil = \left\lceil \frac{3.64}{3.78} \right\rceil = 1$$

El desarrollo de esta aplicación por medio de un solo desarrollador inexperto en el lenguaje, con facilidad de comprensión y aptitudes medias llevaría **3.78 meses**.

El coste dependería del salario del desarrollador. Según http://plandecarrera.infojobs.net/ el salario mensual de un programador medio es de 1370 €/mes por lo que el coste de personal total sería de 5178.6 €.

8. Bibliografía

1. **Gutiérrez., Javier J.** ¿Qué es un framework web? . [En línea] http://www.lsi.us.es/~javierj/investigacion_ficheros/Framework.pdf.

2. Cliente-servidor. [En línea] http://es.wikipedia.org/wiki/Cliente-servidor.

3. **Fernández, Gabriel Marciá.** Universidad de Granada. [En línea] Departamento de teoría de la señal y comuniciaciones, Mayo de 2007. http://0-hera.ugr.es.adrastea.ugr.es/tesisugr/16714763.pdf.

4. **Junta de Andalucía.** Patrón Modelo Vista Controlador. [En línea] http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/122.

5. Modelo Vista Vista Modelo. [En línea] http://www.ecured.cu/index.php/Modelo_Vista_Vista_Modelo.

6. **Mora, David.** Patrón Modelo-Vista-Modelo de Vista (MVVM) Explicado. [En línea] 28 de Mayo de 2010. http://maromasdigitales.net/2010/05/patron-mvvm-explicado/.

7. **Piriz, Fernando Machado.** Una introducción simple al patrón Model View ViewModel para construir aplicaciones Silverlight y Windows Presentation Foundation. [En línea] 06 de Junio de 2010. http://fernandomachadopiriz.com/2010/06/09/una-simple-introduccin-al-patrn-model-view-viewmodel-para-construir-aplicaciones-silverlight-y-windows-presentation-foundation/.

8. **Campos, Oscar.** Patrón de diseño MVC del lado cliente con Backbone.js. [En línea] 17 de Julio de 2011. http://www.genbetadev.com/desarrollo-web/patron-de-diseno-mvc-del-lado-cliente-con-backbonejs.

9. **http://www.lostiemposcambian.com/.** Backbone vs Angular vs Ember. [En línea] 11 de Diciembre de 2013. http://www.lostiemposcambian.com/blog/javascript/backbone-vs-angular-vs-ember/.

10. **Meneses, Rafael.** Estimación de Tamaño de . [En línea] http://tongo.uniandes.edu.co/~csof5101/dokuwiki/lib/exe/fetch.php?media=principal:csof5101_-_estimacionpuntosfuncionales.pdf.

11. **Houston, Koni Thompson.** Evolving Standards in Function Point/Lines of Code Ratios. [En línea] Ocubre de 2003.

http://sunset.usc.edu/events/2003/Presentations/Evolving%20Standards%20in%20Function%20Point.pdf.

12. **QSM.** Function Point Languages Table. [En línea] http://www.qsm.com/resources/function-point-languages-table.